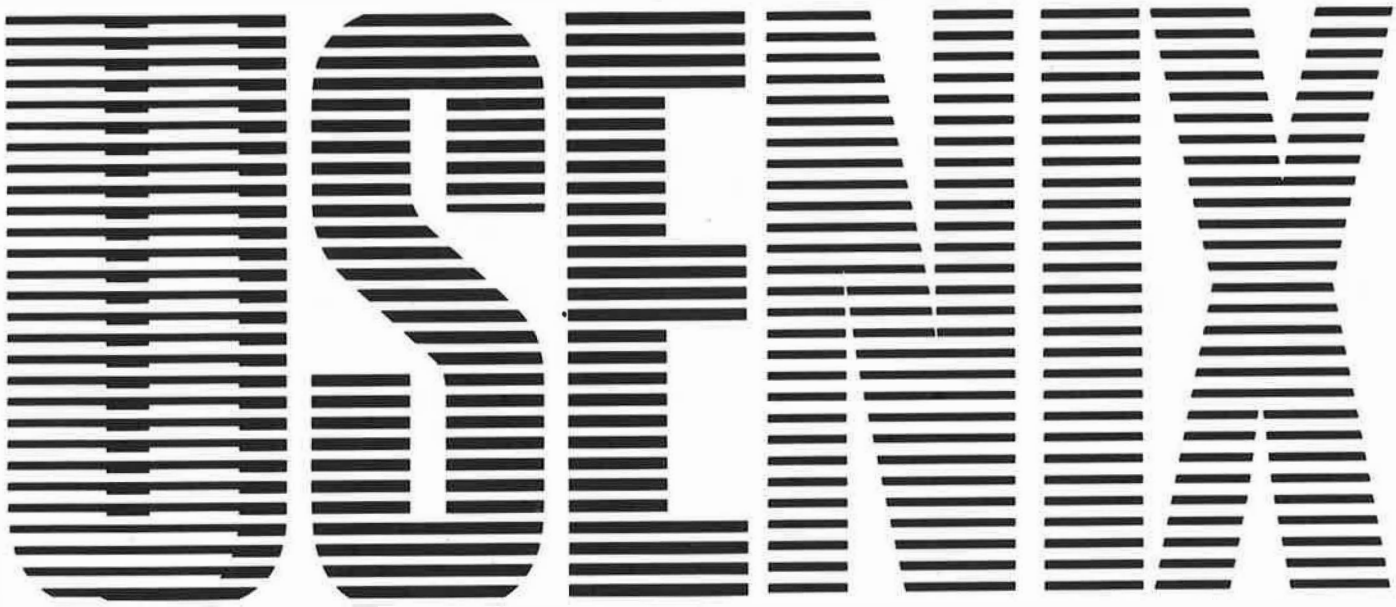

PROCEEDINGS

UNIX Security Workshop

Portland, OR

August 29–30, 1988

The USENIX Association

The USENIX Association is a non-profit association of individuals and institutions interested in fostering innovation and sharing ideas, software, and experience where UNIX and UNIX-like systems and the C programming language are concerned. USENIX sponsors technical conferences and workshops and an annual vendor exhibition; publishes *;login:* (a bi-monthly newsletter) and *Computing Systems* (a technical quarterly); distributes 2.10BSD and the 4.3BSD manuals; and serves as coordinator of a software exchange for appropriately licensed members.

The individual and institutional members of USENIX are interested in problem-solving with a practical bias, with research that works, and with timely responsiveness within a community made up of executives and managers, programmers, and academics.

There is a membership application on the inside back cover.

Future Events

AUUG Spring Conference
Melbourne, Sept. 13-15, 1988

For information, write Tim Roper at *uunet!munnari!labtam.oz!timr*

UNIX and Supercomputers Workshop
Pittsburgh, PA, Sept. 26-27, 1988

Program Chairs are Melinda Shore of the Frederick Cancer Research Facility and Lori Grob of New York University.

EUUG Autumn Conference
Portugal, Oct. 3-7, 1988

C++ Miniconference
Denver, CO, Oct. 17-21, 1988

The Program Chair is Andrew Koenig of AT&T.

Large Installation
Systems Administration II
Monterey, CA, Nov. 17-18, 1988

The Program Chair is Alix Vasilatos of MIT's Project Athena.

USENIX Winter Technical Conference
San Diego, Jan. 30-Feb. 3, 1989

EUUG Spring Conference
Brussels, Apr. 10-14, 1989

Long-term USENIX Conference Schedule

Jan 30-Feb 3	'89 Town & Country Inn, San Diego
Jun 12-16	'89 Hyatt Regency, Baltimore
Jan 22-26	'90 Omni Shoreham, Washington, DC
Jun 11-15	'90 Marriott Hotel, Anaheim
Jan 21-25	'91 Dallas
Jun 10-14	'91 Opryland, Nashville
Jan 20-24	'92 Hilton Square, San Francisco
Jun 8-12	'92 Marriott, San Antonio

Program and Table of Contents

UNIX Security Workshop
August 29-30, 1988

Sunday, August 28, 1988 — Salon I

Registration and Get-Together 7:30 - 10:00 pm

Monday, August 29, 1988 — Salon E

Coffee 8:30 - 9:00 am

Opening Remarks 9:00 - 9:15
Matt Bishop, Dartmouth

Passwords and Authentication 9:30 - 10:30

Break 10:30 - 11:00

Passwords and Authentication (continued) 11:00 - 12:00

Lunch 12:00 - 1:30

Access Controls 1:30 - 3:30

Tuesday, August 30, 1988 — Salon E

Conference Chair:
Matthew A. Bishop
Department of Mathematics and Computer Science
Bradley Hall
Dartmouth College
Hanover, NH 03755
uunet!bear.dartmouth.edu!bishop

USENIX Conference Coordinator:
Judith H. DesHarnais

USENIX Conference Liaison:
Alan G. Nemeth

Preprint Production:
Tom Strong
Ellie Young

NOTES

UNIX Password Security

Lon E. Anderson

Enigma Logic

1. Introduction

Of the many methods for securing a UNIX machine, the vast majority can be broken down into two categories: 1) protection from unauthorized users, and 2) enforcement of internal file security. This paper describes a method of identifying users to the operating system that is different from traditional password systems. This method falls into category 1) above.

In a traditional login security system, the user is identified to the system by what he knows. Unfortunately what a user knows is often compromised. A colleague looking over a shoulder may have gleaned the password. A hacker may have tapped into the phone line and captured the password on its way to the host. A PC may have been used to guess the password for the system. Studies of fixed password systems have shown a dangerous number of user passwords are common names of easily guessed objects. With a dictionary of less than 2,000 words sorted in a most-used to least-used ordering and a selection of typical accounts associated with users whose names are published in a directory, most systems may be broken in one night, sometimes even in minutes.

The problem might not be that bad if it were not for the availability of the 'root' login. If a hacker gets the password for this account, he has free reign on the system, deleting or downloading files at his will. Another problem with UNIX is that the typical UNIX system is left up for 24 hour periods. This then requires that more than one person be given the root password in case there are any problems during the off hours. There is now more visibility for that password, the password must be coordinated in some way (a memo, written on a chalk board), and the system manager has lost true one to one accountability for root.

This has lead to the creation of elaborate schemes to force the changing of passwords based either on the passage of time or a certain number of uses. The basic flaw with these systems is that they rely on individuals. These systems break down because the typical user will write down his password. And if he does not write his password down, there is a high probability that he will forget it. Many people write their passwords down without even knowing it. They use PC programs with menu-driven script replay capabilities to log into UNIX machines. For ease of use, they store their passwords and login IDs in these script programs. Anyone sitting at the keyboard of this machine will find themselves with easy access to all the systems used by the owner of that PC.

2. An Automatic Password Management System

Realizing the problems inherent in fixed password systems, Enigma Logic, Inc. has created a system with automatically changing passwords. The passwords are generated by a small hand-held calculating device. The device has numeric keys

and an issue-password button. Figure 1 shows a typical user login sequence.

```
login: john

SafeWord Security Check  V3.30

Challenge: 2634 2556
PassWord: 1616
(If you wish to change your password, press <ESC>
Fixed password: Eag1046

%
```

Figure 1: A typical user login. (What the user types is in boldface.)

To understand where and how SafeWord is installed on a UNIX system, a description of a user login on a UNIX system will be outlined.

On a UNIX system, users see a prompt "login:" on their screen. This prompt comes from a process called *getty* (which stands for "get tty input"). After *getty* sets the terminal speed and line characteristics, it displays the "login:" prompt. *Getty* then waits for a sequence of characters terminated by a carriage return. When *getty* sees such a sequence, it overlays itself with *login* and passes to *login* the user ID.

login prompts for the fixed password from the user, encrypts it and compares this to the password for this user as found in */etc/passwd*. If the encrypted representations of the passwords match, then the last field of the */etc/passwd* entry for this user is invoked. This is typically the users shell (*/bin/sh*, */bin/csh*, etc.).

On a SafeWord system, the users fixed password field in the */etc/passwd* file is left blank. This is to cause UNIX to not prompt for the fixed password. The shell field of the */etc/passwd* entry for the user is set to */safeword/indent*, which is the software lock. This automatically invokes SafeWord on login. The file */safeword/indent* is set with the following attributes:

```
-rwsr-xr-x  1 root  root  84234 May 4, 1988 10:30  ident
```

The setuid bit causes *ident* to run with root level privileges from a user account. This allows *ident* to access its user database files, which are owned by root.

Referring back to Figure 1, the ID typed by the user is retrieved by *ident*. *ident* then looks for the user in its database files. If the user ID is valid, *ident* then determines the users encryption algorithm. An algorithm is a determinant of the users identity. Each user has a unique algorithm that produces random numbers in a particular and unique way. Given an input to the algorithm of 8 digits, it will output 3 or 4 digits. For a unique input, a particular output will be produced. It is this attribute of the algorithm which makes it possible to authorize users.

ident encrypts the current time and produces a unique 8 digit challenge as in Figure 1. The user types this challenge into his password issuing device and receives an answer on the display. He then types this answer into the terminal and *ident* compares that to what it has already calculated as the correct answer for the claimed user ID. If the answers match, the user is allowed in and his shell is executed. If the answer is wrong, *ident* dies, which causes it to spawn another *getty*

for this tty.

3. The Future

The future of computer access holds many possibilities. Biometrics, the process of determining access based on what you are (retina scan, fingerprint analysis) is coming down in price. New password generating devices are becoming available from various vendors all the time. Enigma Logic currently works with the following password devices: the SafeWord Decoder and Key manufactured by Enigma Logic; the WatchWord by Racal; the SecureNet by Digital Pathways; and the SafeCard by Enigma Logic. Enigma Logic has a commitment to each of these and all emerging technologies.

Enigma Logic was the first company to implement changing password technology. Enigma Logic software supports a wide variety of UNIX and non-UNIX machines ranging from IBM-VTAM, to DEC VAX to any UNIX System III all the way to BSD 4.3 and System V.3. Several PC products are also available to protect UNIX networks that use PCs. Enigma Logic's SafeWord UNIX-Safe is the only UNIX product to date that has received the National Computer Security Center approval as a security add-on product.

A Framework for Password Selection

Ana Maria De Alvaré

E. Eugene Schultz, Jr.

Lawrence Livermore National Laboratory
University of California

ABSTRACT

A major problem in computer security is intrusion into systems due to compromised authentication procedures. This paper will focus on the most commonly used authentication procedure—use of passwords. We have developed a framework for a methodology to estimate the guessability of passwords which, if implemented, could substantially reduce the number of compromised authentication procedures.

Users often invent their own passwords, or are assigned passwords by a system manager. Usually, user-selected passwords consist of potentially highly guessable strings such as one's first or last name, relatives' names, phone numbers, nicknames, or some similar attribute of the user. Passwords selected by system managers are often generated by a simple rule such as user's last name + year or phone extension. When an intruder discovers one password, a rule for generating passwords may be inferred and then used to generate possible passwords, so the probability of intrusion into other accounts increases.

Unfortunately, once a password is chosen, a user is unlikely to change that password until the user discovers intrusions or attempted intrusions into his/her account. A methodology that enables users and system managers to estimate the guessability of passwords would enhance security procedures substantially by enabling them to discard passwords which are highly prone to guessing by intruders. In this paper, we describe a framework for such a methodology.

We assume that user- and system manager-derived passwords are often based on rules. Non-users can be instructed to guess passwords with and without the benefit of information about these rules. Hit rates (the percentage of passwords correctly guessed within a limited number of attempts) are obtained. This method can be used to develop metrics for guessability of classes of passwords.

We ran a pilot study to determine the feasibility of this framework. Twenty-two randomly selected employees of Lawrence Livermore National Laboratory and two employees of NASA Ames Research Center participated in this study. Participants were given 20 attempts to guess a eight-character password which was either a common English word or two unrelated words joined by a control character (eight characters in all). Participants were also either provided with a clue about the password they were trying to guess or were not provided with such a clue. The clue for the common word password was a word from the same conceptual category (i.e., food), whereas the clue for the other password was a word with the identical structure of the password (3-letter word + control character + 4-letter word). Without any clue, no one guessed any password within 20 attempts. When a clue was provided, four of the six people assigned to the

common word password condition were able to guess the password within 20 attempts, but none of the people assigned to the other password condition were able to guess the password within 20 attempts.

The framework we have suggested implies that computer security experts at a particular corporation or institution might conduct guessability studies on a large number of candidate passwords. The results of such studies would remain confidential to the experts who conduct the study and system managers with a strong need to limit intrusions into a system. A system manager might utilize results such as the preliminary results we have obtained by encouraging users to avoid choosing passwords which are closely associated with account names or which have been shown to be highly vulnerable to guessing. Users who have passwords which guessability studies show to be vulnerable to guessing should be encouraged to change their password to one which is rated low in guessability.

Another alternative would be to have computer-generated passwords, which are often similar in structure to the 8-character passwords explored in our preliminary study. Computer-generated passwords are, however, difficult to remember. People tend to write these passwords down, therefore, which leads to other possibilities for password compromise.

Authentication of Unknown Entities on an Insecure Network of Untrusted Workstations

B. Clifford Neuman†, Jennifer G. Steiner

Project Athena
Massachusetts Institute of Technology

ABSTRACT

Project Athena provides computing resources for undergraduate education at MIT.[1] Over 750 computers are scattered across 30 subnets, and support more than 5,000 active users. Single user IBM RT/PCs and DEC VaxStation IIs running versions of the Unix operating system access servers (mostly Vax 11/750s) across the network. Workstations are publicly and privately owned. In both cases the user has complete control over the computer and can easily gain superuser status. Because of this, workstations cannot be trusted to accurately identify their users. The network can't be considered secure either. Users can listen to network traffic as well as generate traffic with forged addresses. Servers are scattered across campus. It is possible that users might be able to physically compromise the security of some of the them.

A method was needed to authenticate users wishing to access network services such as file storage, electronic mail, remote login, and printing. The method had to be secure in the given environment, but not unduly cumbersome for the user. Ideally, the system would appear to the user as if only a single system were being used. Any solution chosen had to scale well. Additionally, compromise of any of the servers could not affect the security of the others.

The approach taken is based on a cryptographic protocol by Needham and Schroeder.[2] An authentication server known as *Kerberos*[3, 4] runs on a trusted computer. Kerberos knows the passwords (encryption keys) for each user under its authority. It also shares a key with each server. When a program running on a workstation (e.g. *rlogin)* wishes to prove the identity of its user to a given network server (e.g. *rlogind)*, it contacts Kerberos and asks for a *ticket* for that server. The ticket is returned to the workstation encrypted in the server's key, and then again in the user's key. The user's password is used to decrypt the ticket which can then be passed to the server to prove the user's identity.

In addition to the ticket, Kerberos generates and returns to the user a temporary encryption key, or *session key*. This, like the ticket, is sealed in the user's password. A copy of the session key is also enclosed in the server ticket. Once the server decrypts the ticket with its key, both the user and server know the session key, which can be used to encrypt further communication between them. In this way, the Kerberos server also acts as a key distribution center.

A ticket can be reused, but additional information passed to the server along with the ticket prevents replays by an imposter. The initial ticket obtained is for a

† Author's present address: B. Clifford Neuman, Department of Computer Science (FR-35), University of Washington, Seattle, Washington 98195.

ticket-granting service which can be used to obtain tickets for other services. In this way, the user only has to enter a password once per login session. Tickets have a finite lifetime, and an attacker who manages to steal tickets from a user can use them for only a short time (relative to the life of the user's password), and only from a particular network address.

Under the Kerberos model, the world is divided into separate domains of authentication authority, called *realms* , each with its own Kerberos server. Principals registered in one realm can easily authenticate themselves to servers in other realms. This is accomplished through ticket-granting servers which are registered in multiple realms.

Kerberos is implemented as a server that runs on a secure machine, and a set of libraries that is used by client applications and services. The initial implementation uses DES for encryption, but encryption is supported in a separate module that is easily replaced.

Kerberos has been in use at MIT for two years, and is currently in beta test at 18 sites across the country. At MIT, Kerberos supports more than 8,000 entities (users and servers) in three different realms. It is used for authentication in rsh, rcp, rlogin, Sun's Network File System, mail, bulletin boards, notification and administrative applications. In summary, Kerberos allows users to authenticate themselves to network services without entering a password at every request, and without relying on less secure methods, such as the host-authenticated *.rhost* mechanism.

References

1. E. Balkovich, S. R. Lerman, and R. P. Parmelee, "Computing in Higher Education: The Athena Experience," *Communications of the ACM*, vol. 28, no. 11, pp. 1214-1224, ACM, November, 1985.

2. R. M. Needham and M. D. Schroeder, "Using Encryption for Authentication in Large Networks of Computers," *Communications of the ACM*, vol. 21, no. 12, pp. 993-999, December, 1978.

3. S. P. Miller, B. C. Neuman, J. I. Schiller, and J. H. Saltzer, *Section E.2.1: Kerberos Authentication and Authorization System*, Project Athena Technical Plan, M.I.T. Project Athena, Cambridge, Massachusetts, December 21, 1987.

4. J. G. Steiner, B. C. Neuman, and J. I. Schiller, "Kerberos: An Authentication Service for Open Network Systems," in *Usenix Conference Proceedings*, pp. 191-202, Dallas, Texas, February, 1988.

CRACK: A Distributed Password Advisor

T. M. Raleigh and R. W. Underwood

Bellcore

ABSTRACT

The Computer Science and Mathematics Research Divisions at Bellcore run a mixed collection of UNIX based superminicomputer and workstations (e.g., VAX 8650's, Convex, Alliant, CCI, SUN, IRIS, etc.), all connected by a hierarchy of local area, wide area and experimental networks. Collaboration with outside university researchers, visiting researchers, contractors and summer graduate students produces a large community of users with diverse backgrounds and attitudes toward security. CRACK is an attempt to harden the passwords in use by providing a network service that attempts to break passwords against common usage libraries and then challenge users to supply new ones. The system provides a facility to accumulate passwords that have been cracked and to accept donated passwords but makes no attempt to maintain records correlating users with their passwords.

The system is implemented as a network service that receives requests to crack passwords and responds with an indication that the password is crackable (not with the password itself). Subscribers to the service submit a request containing the encrypted password, the type of encryption algorithm used (e.g., DES), a code indicating the qualifications placed on the password (e.g., greater than 6 characters, minimum of 2 numbers, etc.), a level of cracking to be attempted and the priority of the attempt. Levels of cracking are defined by the server and correspond to a sequence of libraries and permutations of those libraries that are used in cracking passwords. Only the administrator of the advisor can create new levels and any associated libraries. The advisor responds to the requesting system via a callback mechanism that indicates that the password has not been broken or if it has, the amount of time taken, the processor class and the library used to break the password are given. Systems subscribing to the advisor notify users whose passwords have been broken to change them when they login again.

Priorities are assigned to requests since changing the root password is a significant event and should take precedence over routine user passwords. The priority mechanism also allows the introduction of parallelism into the advisor. A list of cooperating slave systems and the library configurations on each slave is kept by the advisor which optionally can request cracking to occur in parallel on slave systems and/or accelerated cracking on faster systems.

The advisor should eventually become the repository of common jargon, slang, abbreviations (and their permutations) of an organization since it remembers passwords that it has cracked and builds a separate library which is checked against all requests. A mechanism also exists which allows users to contribute their old passwords to the advisor to build a repository of passwords. To encourage this, no records of contributors or users whose passwords are cracked is kept.

The philosophy behind the design is that users will respond to the challenge and feedback of a machine breaking their password by creating more difficult passwords and that the advisor can be easily tailored on a site or user group basis. New

search libraries distilled from user documentation and organization charts are being experimented with for their usefulness and a facility for reporting intermediate results (useful when large libraries are being used) will be added.

A graphics interface is available to display the degree to which cracking has been executed on the user population, the age of passwords and other interesting statistical measures.

UNIX GUARDIANS: Delegating Security to the User

George I. Davida
Brian J. Matt
Electrical Engineering and Computer Science Department
University of Wisconsin-Milwaukee
Milwaukee, WI 53201

August 1, 1988

Abstract

This paper describes the design of a system that allows users to create sophisticated protection environments for their files. The system supports a special class of processes called Guardians. These Guardians change normally passive files into active objects. The file directory tree is partitioned. Subtrees are split off from the main tree and placed under the control of Guardians. The role of "super users" is reduced and file control is distributed back to the users. The system itself is distributed in nature; multiple processors are used to physically separate user processes from the main operating system kernel. Cryptographic facilities are available to protect data, both in storage and in transit through the system.

1 Introduction

Many widely available operating systems provide security by at the cost of maximum user productivity.

> Many commercially available operating systems make it essentially impossible to create or install any user software or application software without administrative help; some other systems make it virtually impossible to read files belonging to another user, even when the users want to cooperate in their work. All these measures work by restricting access to the system and by reducing the powers that the system gives its users. The UNIX system was designed to increase, not decrease, the power and flexibility available to its users[15].

This system was conceived to explore mechanisms to increase the "power and flexibility" of users and administrators in the security realm. This system provides the necessary tools, to allow users to encapsulate critical files within individually tailored, protection environments.

These environments are provided by special processes called Guardians[10]. It is these processes rather than functions inside the "kernel" that ultimately mediate access. With processes providing the interface, access no longer need be a simple matter of "yes" or "no". The Guardian can change its behavior based on time, vary the contents of the file from one user to another, engage in additional authentication procedures and so on.

The system is a modified version of Berkeley 4.2BSD UNIX[1]. The reader is assumed to be familiar with UNIX operating systems. For further information on UNIX see [24,4,17,23,26].

In Gligor et al. secure UNIX systems are categorized into the emulation approach and the restructuring/enhancement approach [13]. In the emulation approach a UNIX emulator and possibly trusted processes are combined with a security kernel to provide a UNIX environment. The use of a security kernel facilitates the use of formal verification techniques but results in some performance penalty. These systems include KSOS[5,21] and UCLA Secure UNIX[22]. The restructuring/enhancement approach involves altering the UNIX kernel and system processes. Traditional UNIX security features are retained and new capabilities are inserted to enhance system security. LINUS IV[18], Secure Xenix[2][13], Gould UTX/32S[3][20] and this system are examples of restructuring/enhancement systems.

A prototype of the system is near completion, it will be operational third quarter 1988.

2 Overview

Our perspective of operating system security is analogous to a building. The frame of the building, its structural support and walls are provided by physical separation, logical separation (the operating system) and cryptography. The rooms of the building are its resources, UNIX files. The occupants of the various rooms (or the Guards at the doors) decide who enters a room and what operations they can perform.

By providing cryptographic functions to the users the operating system's role in data protection can be reduced. Key exposure is an ever present threat. The threat is minimized in this system by storing keys generated by processes beyond the operating system's reach and by the fact that keys can be generated that the operating system can never see. A special hardware board, the Network and

[1]UNIX is a trademark of AT&T Bell Laboratories.
[2]Xenix is a trademark of Microsoft Inc.
[3]UTX/32S is a trademark of Gould Inc.

Security Auxiliary Board (NSAB), is used to provide these features. The NSAB provides cryptographic functions utilizing a DES[2,1] chip. The memory of the NSAB is hidden from the rest of the system and is used to store keys. Other cryptographic routines, including RSA[25], are available in software libraries.

The physical and logical structure of the system combine to isolate user processes. User processes, both Guardian and normal processes, run on slave kernel(s). Only special system processes use the main operating system kernel. By placing user supplied and controlled processes on the slave processor(s) the main operating system is better protected. The reason for this is main kernel never has to relinquish control of its processor to, or share memory with, a user process. Connecting the slave processor(s) to the master processor is a 'local network' (link layer) implemented on the NSAB.

The main kernel is the only processor providing access to the peripheral devices. The master kernel provides virtual memory support for the slave processor(s) along with all file system and device access. Slave kernels convert system calls and other operating system functions into messages for the master kernel requesting necessary services. The master kernel fulfills valid requests and sends the results to the slave kernels.

Having a distinct Guardian for every critical file was not considered desirable. The overhead would be too high. To further reduce overhead and increase compatibility, the system should provide environments for individuals where processes could operate without the need of constant Guardian intervention. In addition, files that share common ownership and function and can be protected by a single Guardian. Therefore, it was decided to detach the sub-trees of the UNIX file system from the root file system and attach them to Guardians.

3 Guardians

Guardian processes alter our concept of a file. The conventional way of perceiving files is a collection of data and some standard set of operating system supplied functions, i.e. *open, read, close*. Since the file owner, by using Guardians, can alter the behavior of these functions and create new ones the files are now object oriented[14].

Guardians have the following structure: the i-nodes[24] of three "files" located on one or more logical devices are "linked" together. This configuration is called a Triad. The three files are designated the Guardian File, the Secure File and the Control File. The executable version of the Guardian itself is stored in the Guardian File. The function of the Control File is left to the designer of the Guardian program; it normally contains access control information for use by the Guardian process. The Secure File typically contains the data or executable being protected.

When a Guardian is executed, the three files of the Triad are opened for the Guardian process and are connected to the three highest file descriptors. Child

processes can inherit any open Triad file but pathname based system calls, like *open* and *exec*, do not function. This is a consequence of the fact that the files of the Triad are not listed in any directory. Without Guardian intervention, the only operations that can be performed on Triads are Triad creation and Guardian File execution and Secure File execution.

Guardians fall into one of three classes by function:

1. File Guardians which protect files during normal file operations like *open, read, write,* etc.

2. Line Guardians which protect home directories and interact with users during login sessions.

3. System Guardians, like the Spawners, that provide system services.

Processes communicate with both types through sockets[19,23].

An advantage of Guardians is that authentication of the entity attempting to use a file is controlled by the owner of the file. For some applications, reliance on the UNIX password mechanism, or whatever method of authentication the user's Line Guardian normally uses, may be unacceptable. In such instances the file's Guardian can request additional passwords from the user or some other identification technique. This can be accomplished by the file's Guardian contacting the user via the user's Line Guardian.

When the system is booted each slave processor begins to execute two special Guardians called the Line Spawner and the File Spawner. These Guardians are actually server server processes, each spawner executing Guardians of its type. The Line Spawner executes Line Guardians as part of the "login process", see section 5, while the File Spawner executes File Guardians for "normal" file access.

Interaction between a requesting process and a File Guardian is via messages. The communication protocol is similar to XEROX Courier[3,9]. Guardians are not executed directly, and as a result there is no inheritance by the Guardian of properties of the user's execution environment. This, and the fact that the Guardians cannot be modified without the cooperation of the Guardian itself, avoids several security problems with UNIX, in particular with *setuid* programs[7,6,26,13,15,16].

3.1 Detached Directory Trees

A Detached Directory Tree (DDT) is a subtree of the UNIX directory system. It is separated from the directory system and connected a Guardian's Triad. When the Guardian is invoked its root and current directory are changed to the root of the DDT[4].

[4]Depending on how the system is configured, the change of directory occurs at Guardian execution or by the Guardian executing a system call.

A typical user possesses a DDT that contains his/her home directory and standard system utilities. This DDT is connected to the user's Line Guardian and becomes accessible as part of the login process, see section 5.

The DDT mechanism is designed as much to keep processes out of a DDT as to keep them in. The Restricted Environments of UTX/32S[20] and Wood and Kochan[26] were developed to contain processes within regions/domains. Also, unlike the Restricted Environments, interprocess communication between processes in different DDT's is not restricted by the system.

The root file system contains a minimal set of files. Most of the normal system files are contained in a DDT. A process's view of the directory system (its sub-view) consists of the files on its DDT, the names of Guardians and names supported by those Guardians.

The normal UNIX mechanism (in the kernel) for mapping pathnames to files is useless when applied to Guardians and therefore so are the pathname based UNIX system calls, including the execution system call *execve*. A new system call is provided to perform the Guardian executions but a translation must be performed by the proper spawner. A requesting process references a Triad by a Triad name, typically in the form of a UNIX pathname, and the spawner converts it into a Triad identifier. The Triad identifier has a one-to-one mapping to the Guardian File on disk. The owner of the Triad must register a Triad with the appropriate spawner prior to its use. The registration can be performed automatically but is currently handled by the system administrator manually. The security of the system does not depend on the Triad identifier being secret.

4 Roots and Setuid

When the system is in secure mode it rejects "super users" and does not allow normal processes to change their ownership. The only changes in process ownership that do occur is when a Guardian is executed. If (somehow) a "super user" process attempts to make a system call, the call is rejected and the "Security Operator" is notified. If by some means a process becomes a root process during a system call, it is trapped when it tries to return from the call and the "Security Operator" is notified.

The usual means for changing the ownership of a process is the execution of a *setuid* or *setgid* file[24]. An additional means of changing ownership available to "super users" are the *setreuid* and *setregid* system calls[17]. The *setuid* and *setgid* bits are ignored while the system is in secure mode and the system calls become null operations.

5 Login Sequence

The User Interface system is a user's first contact with the system. The interface is actually three different programs. A window manager, called the Interface Program, and two others, the Remote Connection Control Program (RCCP) and the Secure Control Channel Program (SCCP). The RCCP and SCCP provide the network communications from the master processor, where the User Interface operates to processes on their slave(s).

The RCCP is used to connect to the Login Spawner on the appropriate slave processor. That Login Spawner passes off the connection to the requested Line Guardian. Once the login sequence is finished, the RCCP talks to the Line Guardian passing messages between the Interface Program and ordinary processes, via the Line Guardian. The SCCP handles security related communications between the Interface Program and the Line Guardian. These messages include those that change the behavior of the Line Guardian, or provide for private communications between the user and File Guardians. The Interface Program utilizes an attention key to control operations like switching current window or adjusting window size. This key also prevents users from spoofing other users with fake login sequences.

A sophisticated Line Guardian can perform continuous supervision of the user. The possibility certainly exists that the identifying property checked at login time (the knowledge of the password) may have been acquired by an imposter. Such a Line Guardian requires a model/profile of the user's behavior perhaps along the lines of Denning's Intrusion-Detection Model [12]. Another possibility for the Line Guardian is to provide multiple subsystems at different security levels[8] depending on the Line Guardian's evaluation of the user. Also the Line Guardian can check incoming messages for bombs.

6 Key Management

In addition to its networking function the NSAB supports cryptographic functions. The NSAB contains set of key registers that are allocated on demand to requesting processes. One set per process. Each key register set can contain up to one public key buffer and up to twenty conventional key buffers. The public key buffer places no constraint other than size on the type of public key algorithm for which it is used[5]. The conventional key buffers are used with the DES chip and are eight bytes in size.

All data and keys involved in request to the NSAB are moved between the the process's address space and the private memory of the NSAB. Each key register set contains information on the process it is associated with to prevent misuse.

[5] The public key buffers don't have to be used for public key cryptosystems at all

Keys "flow" through the NSAB in the following manner. Conventional keys are generated by processes and internally by the NSAB. Internally generated keys are never seen by the UNIX kernels or the processes. The public keys are provided by the processes and can be read from the NSAB. This is accomplished by specifying the key register set address. The address is set by the key register set's owner and cannot be changed until the register set is freed. The addresses are retained to prevent one process from masquerading as another by duplicating an address.

Key registers can share a NSAB generated conventional key, allowing two processes to engage in cryptographically protected communications. A process can dispose of a key buffer at any time; a shared buffer is dispossed when both processes have freed it. When a process exits or requests that its key register be freed, the connected key buffers are disposed of as described above.

The NSAB provides encryption/decryption functions using a DES chip. These functions include the conventional DES operations Electronic Code Book, Cipher Feedback, and Cipher Block Chaining[11], and DES generated pseudo-random pads. The purpose of the the pseudo-random pads is to provide processes sharing the same key the ability to generate synchronized almost random one time pads. These processes thereby avoid making repeated requests to the NSAB for encryptions of small amounts of data.

7 Additional Features

The system has additional features for security enhancement. Each UNIX system has a special Guardian called a Frisker that examines processes. The main UNIX supports a supervisor process which displays system status data and handles audit trial storage.

7.1 Frisking

Frisking involves selecting part(s) of a processes address space and hashing the contents. The result is compared against a precalculated value. The goal is to protect against process tampering and to aid in process identification. A process can request that a frisking Guardian frisk another process if the following hold:

1. The selected Frisking Guardian and the process to be frisked run on the same processor.

2. The requesting process and the process to be frisked share a key buffer.

7.2 Monitoring the System and Audit Trials

The System Monitoring process is executed by the master processor. The System Monitor controls the system's tape drive and audit trial information on

the tape. The audit trial information is provided by processes or the "Security Operator", the individual operating the System Monitor. The Security Monitor also displays messages sent to the monitor by processes. The messages are checked for "bombs" prior to displaying them on the screen. The "Security Operator" uses the system console and the Security Monitor to maintain the system. For example the System Monitor examines a data structure (the security state vector) a copy of which is located on each processor, for tampering, and the "Security Operator" can shut the system down if a problem occurs.

8 Compatibility with UNIX

In order to maintain compatibility with UNIX, not only do user DDT's have to look like mini UNIX file systems but an addition to the UNIX C language library was necessary. To communicate with the File Guardians/and File Spawner(s), message packets are transmitted between a requesting process and the Spawner / Guardian. These messages are handled (on the process's end) by the communication library an addition to the C library. The communication library differentiates system calls made by the rest of the process into two classes, normal and directed at File Guardians. The normal calls do not require special treatment but those that require File Guardian interaction, whether the call has a pathname argument or not, result in a message(s) being sent to either a File Spawner or File Guardian. The goal is that UNIX utilities need only be recompiled with the new library[6].

9 Observations

The system provides a powerful environment for the development of access control and authentication mechanisms. The environments can be tailored to the needs of the individual and are under the control of individual users. However, control over creation of Guardians and their contents could be used to change this discretionary access control system into a mandatory access control system. The system provides a high degree of compatibility with 4.2BSD UNIX. What differences that do exist (except for whatever additional passwords may be necessary) can be made quite transparent to the ordinary user.

References

[1] *The Federal Register*, (149), August 1, 1975.

[6] Actually all that is necessary is to link the object modules of a program with the new C-library (which contains the new system call library) given the object modules were generated for 4.2BSD on this machine.

[2] *The Federal Register*, (52), March 17, 1975.

[3] Courier: the remote procedure call protocol. XEROX Corp., December 1981. Xerox System Integration Standard XSIS 038112.

[4] Maurice J. Bach. *The Design of the UNIX Operating System.* Prentice-Hall, Englewood Cliffs, N.J., 1986.

[5] T. A. Berson and G. L. Barksdale. KSOS-development methodology for a secure operating system. *National Computer Conference, 1979*, 48:365–371, 1979.

[6] Matt Bishop. How to write a setuid program. *;login:*, 12(1), January/Feburary 1987.

[7] Matt Bishop. Security problems with the UNIX operating system. January 1983. Department of Computer Science, Purdue University, (unpublished private communication).

[8] Richard Botting. Novel security techniques for on-line systems. *Communication of the ACM*, 29(5):416–417, 1986.

[9] Eric C. Cooper. *Writing Distributed Programs with Courier.* Technical Report, University of California, Berkeley, March 1982.

[10] George I. Davida and Brian J. Matt. Crypto-secure operating systems. *National Computer Conference, 1985*, 54:575–581, 1985.

[11] Dorothy E. Denning. *Cryptography and Data Security.* Addison-Wesley, Reading Mass., 1981.

[12] Dorothy E. Denning. An intrusion-detection model. *IEEE Transactions on Software Engineering*, SE-13(2), Feburary 1987.

[13] V. D. Gligor, E. L. Burch, C. S. Chandersekaran, L. J. Dotterer, M. S. Hecht, W. D. Jiang, G. L. Luckenbaugh, and N. Vasudevan. On the design and the implementation of secure Xenix workstations. *Proceeding of the 1986 IEEE Symposium on Security and Privacy*, April 1986.

[14] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, Reading Mass., 1983.

[15] F. T. Grampp and R. H. Morris. UNIX operating system security. *Bell System Technical Journal*, 63(8):1649–1672, 1984.

[16] Carole B. Hogan. Protection imperfect: the security of some computing environments. *Operating Systems Review*, 22(3):7–27, July 1988.

[17] William Joy, Eric Cooper, Robert Fabry, Samuel Leffler, Kirk McKusick, and David Mosher. *4.2BSD System Manual, revised July, 1983.* Technical Report, University of California, Berkeley, 1983.

[18] Steven Kramer. Linus IV - a experiment in computer security. *Proceeding of the 1984 IEEE Symposium on Security and Privacy*, April 1984.

[19] Samuel J. Leffler, Robert S. Fabry, and William N. Joy. *A 4.2BSD Interprocess Communication Primer, DRAFT of July 27, 1983.* Technical Report, University of California, Berkeley, 1983.

[20] Greta Miller, Steve Sutton, Mikel Matthews, Joanna Yip, and Tim Thomas. Integrity mechanisms in a secure UNIX: Gould UTX/32S. *Second Aerospace Computer Security Applications Conference*, 48:19–26, 1986.

[21] T. Perrine, J. Codd, and B. Hardy. An overview of the kernelized secure operating system (KSOS). *Proceedings of the 7th National Computer Security Conference*, 146–160, 1979.

[22] Gerald J. Popek, Mark Kampe, Charles S. Kline, Allen Stoughton, Michael Urban, and Evelyn J. Walton. UCLA secure UNIX. *National Computer Conference, 1979*, 48:355–364, 1979.

[23] John S. Quarterman, Abraham Silberschatz, and James L. Peterson. 4.2bsd and 4.3bsd as examples of the UNIX system. *ACM Computing Surveys*, 1985.

[24] Dennis M. Ritchie and K. Thompson. The UNIX time-sharing system. *Bell System Technical Journal*, 57(6):1905–1929, 1978.

[25] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptograms. *Communication of the ACM*, 21(2), Feburary 1978.

[26] P. H. Wood and S. G. Kochan. *UNIX System Security.* Hayden, Hasbrouck Heights, N.J., 1984.

Multilevel Security with Fewer Fetters‡

M. D. McIlroy

J. A. Reeds

AT&T Bell Laboratories

ABSTRACT

We have built an experimental UNIX† system that provides security labels (document classifications), where the security labels are calculated dynamically at the granularity of kernel activity, namely, at each data transfer between files and processes. Labels follow data through the system and maintain the lowest possible classification level consistent with the requirement that the labels of outputs dominate the labels of inputs from which they were computed. More rigid control is exerted over the labels of data passing out of reach of the system to and from tapes, communication lines, terminals, and the like. Necessary exceptions to the security rules (as for system administration, user authentication, or document declassification) are handled by a simple, but general, privilege mechanism that can restrict the exceptions to trusted programs run by "licensed" users. Privileges are subdivided; there is no omnipotent superuser. Carefully arranged data structures and checking algorithms accomplish this fine-grained security control at a cost of only a few percent in running time.

Dynamic labels should help mitigate the suffocating tendencies of multilevel security. At the same time dynamic labels admit covert channels by which dishonest, but authorized, users can leak data to unauthorized places at modest rates. The system is still highly resistant to other kinds of threat: intrusion, corruption of data by unauthorized users, Trojan horses, administrative mistakes, and joyriding superusers. In most real settings, we believe, worries about potential leaks will be far outweighed by these latter concerns and by the overriding consideration of utility.

The standard security mechanisms of the UNIX system are, in military parlance, "discretionary": protection depends primarily upon the individual owners of data taking care to set permissions on files. Some automatic help is offered: the owner/group mechanism, *umask*, and clean files uncontaminated by old trash from shared disks. The responsibility for further precautions, such as setting owner-only permissions on files in the shared temporary directory, is delegated to

‡ Presented at EUUG, London, April, 1988.

† UNIX is a trademark of Bell Laboratories.

programs.

A carefully administered UNIX system can be quite resistant to penetration. But careful administration is not easy. Great reliance is placed on the probity, accuracy, and vigilance of superusers. It is all too easy for a busy superuser inadvertently to misset permission bits, to execute a Trojan horse[1], to make temporarily unprotected copies of secrets, or to promote unvetted files to trusted status.

Because UNIX systems are simple enough to be administered by amateurs, whose first interest is in use, not operation, security holes are the rule, not the exception, in real life. A few famous security holes have been distributed in major software. And some add-ons, such as Berkeley's network file system, seem to have been deliberately designed for insecurity.

Is there, then, hope of running a UNIX system securely? Of course. Lots of systems are run quite soundly today. Raising the question a notch, is there hope of making a UNIX system that conforms to government-style security policies?

In a sense the answer is yes; UNIX systems need less fundamental patching than most to achieve government security goals. But the yes must be qualified. In almost any production system it is difficult, if not impossible, to guarantee that many megabytes of code do not contain one fatal crack that can topple the whole edifice. Such guarantees are especially elusive in environments that change frequently, as do so many typical UNIX installations. How can one be sure that no flaws are inserted by any of the software tools—shell, editor, compiler, assembler, library—that touch new code as it is being installed?

In another sense the answer is not so clear. Rigid security measures must surely damage the plasticity that attracts users to the UNIX system. Is the conflict so fundamental that a "secured" system will lose its appeal? The possibility is very real under ordinary security models; hence we have undertaken a somewhat more flexible approach.

We have built an experimental system with mandatory controls: security classification automatically follows data through the system. At the same time we have blunted the vulnerability of the system to mistakes by the superuser. In our system all data files have security classification *labels*, with "higher" labels designating more sensitive data. The normal flow of data must be *up*: in general output labels must be at least as high as input labels. A security system also needs escape hatches for declassifying data that is no longer sensitive, or extracting nonsensitive parts from sensitive documents. For this purpose we allow certain carefully designed *trusted* programs to violate the rules and produce output with labels lower than input.

We have attempted to provide mandatory, inescapable controls without utterly destroying the basic feel and productivity of the system. To obtain early warnings of snags caused by the controls, we are doing our development work

[1] The hoariest of all: a bad guy feigns trouble, and asks a superuser for help. The first thing the superuser does is *cd badguy; ls*. The game is up. The bad guy's own program named *ls* has been executed by the superuser; it has silently bugged some setuid root program, removed itself, and then called the real *ls*.

under the system itself. At the time of writing we are working quite honestly and relatively comfortably within the confines of the system.

The main idea

Each file or process has a label, shared by all data in it[2]. Terminals and other devices such as tapes have labels that reflect the system's understanding of the clearance of the source to which the device is currently connected. The labels form—almost—a mathematical lattice. Whenever a system call causes a transfer of data, the labels are checked to ensure that data flows only up the lattice.

The security of data explicitly passed among labeled entities is safeguarded. Examples of protected transfers are bytes transmitted by *read* and *write* and bits set by *chmod*. Implicitly set inode data, such as file modification times and link counts, are also protected as far as possible without making the system unusable.

Other ways of communicating information, including but not limited to arguments of *exec*, error returns from system calls, file access times, the identity of open files, and otherwise inferred knowledge, we declare to be "covert channels." We have studied covert channels and arranged to throttle or stop completely covert channels of significant bandwidth. In effect we have divided information transfers into "lawful" transfers, which honor the US Department of Defense "Orange Book"[3], and covert channels. Just which covert channels to leave unplugged we have decided by balancing risk versus utility and compatibility.

We keep all processes and files at their minimum allowable labels as long as possible. The label of a process will increase only when necessary and only as far as needed to allow reading of inputs. Similarly when the label of a process exceeds the labels of its output files, the file labels will rise.

A few system programs must be exempt from the usual label checking. Such programs are granted special privileges—for instance to set the label on a user's terminal at login time, to read foreign tapes, or to perform backups. These privileges are zealously guarded: no program can pass its privileges intact to another or alter a privileged program in any way (aside from removal of privilege).

Thus we have three kinds of security mechanism in our system: (1) the usual discretionary permission scheme, based on userid and groupid and the familiar *rwxrwxrwx* bits, but with the superuser stripped of the right to ignore permissions, (2) the mandatory label scheme, which strives only to maintain correct label relationships, and which pays no attention at all to userid or groupid, not even superuser, and (3) the privilege scheme, which guards the administration of labels and of the privilege scheme itself.

---

[2] For technical reasons, seek pointers also have labels of their own. Seek pointers are shared between processes; information can flow through a shared seek pointer (via *lseek*) at a substantial rate—thousands of bits per second. Since a seek pointer is "written" into by *read* as well as by *write*, the contents of a seek pointer, unlike the regular contents of a file open for reading, must have a label as high as that of the reading process. Hence the separate label.

[3] *Department of Defense Trusted Computer System Evaluation Criteria*, Department of Defense Computer Security Center, Fort Meade, MD, 15 August 1983. This bible has set the terms of discussion for most current work in computer security.

Labels

A label can be any element of a given finite lattice. In addition there are two nonlattice labels, **NO** and **YES**. No data may flow to or from a file or inode labeled **NO**; it is effectively blocked out of the system, and can only be readmitted by special arrangement. "External media," such as terminals, tape drives, and raw disks, where labeling is beyond the control of the usual mechanism, are normally marked **NO**. Label **YES**, on the other hand, is universally permissive. Only one file, */dev/null*, is marked **YES**. At the moment no other file can gain such blessing, but it might also be appropriate for an append-only audit file.

Our lattice is the lattice of subsets of 480 items, represented by 60-byte bit vectors. How these bits are used is arbitrary. For example, the first three bits might represent the customary classification levels—unclassified, confidential, secret, top secret—encoded as 000, 001, 011, 111 respectively. Further bits might represent compartments: 000 100 for Iran, 000 010 for Nicaragua, etc. Oliver North would have been cleared for 111 110. A possible history of a process initially labeled secret (011 000) is:

> Create a new file *north/contragate*; it is labeled (000 000) by default, but writing in directory *north* causes the label of *north* to become at least secret (011 000).
> Read *iran.data*, which, say, is confidential and compartmented (001 100). The process label rises to (011 000)∪(001 100) = (011 100).
> Read *nicaragua.data*, top secret and compartmented (111 010). The process label rises again to (111 110).
> Write *north/contragate*. The file label rises to (111 110). The directory label is unchanged.

Not all labels can change automatically. A label may be "frozen", which stops operations that would normally require a label change. In particular, labels of terminals are guaranteed to be frozen, typically at the value determined by login. Suppose our example process had been initiated from a terminal that had been cleared only for top secret Iran data (111 100) and attempted finally to write to the terminal. The write would fail, thus keeping Nicaragua data from a user not known to be cleared for it. Further attempts to launder the label, perhaps through a pipeline like *cat north/contragate | grep .*, would meet the same fate. Only a properly authenticated fresh login (or subsession) can authorize the terminal for the higher label.

The idea of a lattice of labels is well known. Our deviation from the strict model, with **NO** and **YES**, answers needs to regulate entry from places where labels are not under control of the system, and to deal with the important special case of */dev/null*.

Privileges

Our privilege mechanism is simple, but flexible. In its purest form it restricts special powers to trusted users using trusted tools.

To some extent the privilege mechanism may be understood as partitioning the supreme powers once accorded to the superuser. Superuser status itself is diminished. The superuser is fully bound by security labels and cannot ignore write

permissions. Largely to avoid rewriting masses of code, the superuser retains most other powers. Thus the superuser can still do damage (to data he is cleared for), but mainly by tedious methods that leave tracks—changing modes and owners. Superuser status must be augmented by privilege to execute powerful restricted system calls such as setting the userid or mounting a file system.

We have identified five distinct privileges, listed below, each governed by one-bit *licenses* and *capabilities*, which are separate from labels. A trusted process or file is one with some nonzero capability or license. In the strictest policy regime each privilege of a process p executing file f is determined by the intersection of the process's license for that privilege and the file's capability for the same privilege:

$$Priv(p) = Lic(p) \cap Cap(f).$$

Process licenses are assigned at login, are inherited across *exec*, and may be relinquished at will, never to be regained. Licenses effectively identify trusted users, while capabilities identify trusted programs.

The trusted-user-trusted-tool model of privilege may be eased in various ways. It is possible to grant a default "system capability", $Cap(s)$, to every file by the rule

$$Priv(p) = Lic(p) \cap (Cap(f) \cup Cap(s)).$$

By setting $Cap(s) =$ **true**, we can make $Priv(p) = Lic(p)$, which means that any program can do magic provided its user is licensed. In such a regime a superuser possessing licenses for all privileges could act with the same impunity as a standard superuser.

It is also possible to give a program file a license, $Lic(f)$, making the program "self-licensing" for one or more of its capabilities. Then the effective license of a process p executing program f is $Lic(p) \cup Lic(f)$. Self-licensing is limited by another policy constant, the "system license", $Lic(s)$, which is used as a mask. The full formula for determining each privilege of a process is

$$Priv(p) = (Lic(p) \cup (Lic(f) \cap Lic(s))) \cap (Cap(f) \cup Cap(s)).$$

In a typical self-licensing case, where $Cap(s) =$ **false**, $Lic(f) = Cap(f)$, and $Lic(s) =$ **true**, this reduces to $Priv(p) = Cap(f)$. In this regime a self-licensed program gets power in much the same way as does a setuid-root program in standard systems, except that the power is not inherited across *exec*.

With appropriate settings of the two (compile-time) system policy constants, $Lic(s)$ and $Cap(s)$, our privilege model is able to mimic the disparate privilege features of most current operating systems. In our experimental system we have set $Cap(s) =$ **false** for every privilege. We have also also set $Lic(s) =$ **false** for the most powerful privilege, "set privileges". Thus privileges can be set only by trusted users using trusted programs.

The five privileges are:

Mount. The right to make new data sources or sinks available to the system. One way is by changing a file label away from **NO**; a second is by the *mount* system call; a third is by changing the label on an external medium. A process with mount privilege would normally execute an authentication protocol before actually

performing any of these operations.

Nocheck. The right to read or write data without regard to security label (but still respecting the standard permission scheme). Although mount and nocheck both provide extraordinary access to data, they are qualitatively different. Nocheck handles (and may censor) every suspect bit. Mount opens resources to the whole system—a much more sensitive responsibility.

Set licenses. The right to increase the license or ceiling of a process. The principle use for this is in setting up "sessions", where a user entitled to play more than one role wishes to suspend one role temporarily and switch to another. Sessions are merely a refinement of *su*, which changes rights by the crude expedient of changing identity.

Set privileges. The right to change file capabilities and licenses. We expect not more than one or two programs to be given this most powerful of all capabilities. In a thoroughly security-conscious installation, only an identified security administrator, different from the system administrator, would be licensed to set privilege.

Write uarea. The right to change values, such as userid, that are remembered by the system for the benefit of the process and its offspring. This peculiar capability arises because a child process need not be as highly classified as its parent. Without some control, uarea items (especially BSD group permissions) would provide a covert channel of significant bandwidth.

By dividing privileges we promote safety from errors by an omnipotent superuser. At the same time we introduce complexity, which can cut the other way. Thus we have deliberately kept the number of identified privileges small. We have refrained from defining new special roles (for example system administrator, operator, or security administrator) in the superuser tradition. Notions of such roles did influence our choice of privileges and will guide the design of administrators' tools. But the notions seem inappropriate to build in at the ground level: no single administrative model makes sense across the spectrum of real installations.

System features

To implement the above facilities relatively few new system features are involved:

New system calls get and set file labels. Another new system call sets the process label. Privileges and frozenness are set along with labels. Unless executed by a trusted process, the system calls permit only safe changes: labels may not decrease; process privileges may not increase; file privileges may not be changed.

A special system call allows nocheck processes to confine their powers to certain files. For example, consider *df*, which needs nocheck privilege to read the file system device. Its outputs, however, should be subject to ordinary security checks to prevent a mole from getting his message through[4].

[4] If *df* is exempt from all security checks, the mole can get a message to the standard output this way:

```
df /dev/disk0 /dev/disk1 /dev/disk1 >unclassified
```

Every process has an inherited *ceiling* label, above which the process cannot do any business. This has little to do with stopping ordinary leaks: if a lowly process raises its label high, its output will be high and thus protected anyway. It does, however, cut off some possibilities for mischief with covert channels. And it prevents unauthorized userids from injecting noise in high places.

Mounted file systems also have ceilings, both on labels and privileges. File system ceilings may be used to restrict the content of file systems being prepared for export, or to prevent contamination, especially by unknown privileged files, from imported file systems.

A directory may be "blinded." Blind directories are immune to automatic label changes and thus provide a convenient way to gather, yet keep hidden, data of disparate labels, as for the temporary directory */tmp*. Untrusted processes cannot open a blind directory for reading, and every new file created in such a directory is assigned a random name. A new system call retrieves the name.

Implementation

Dynamic label changing involves considerable overhead of implementation. It is insufficient simply to add label checks at file open. In principle, labels must be checked on every read, every directory search, and every write, including writes of new entries into directories. When a write check fails, the file label is raised if possible; for a read the process label is raised. Every other process dealing with the file must become aware of the change on a fine time scale; in the worst case a label may change between disk blocks of a long IO transaction. A carefully designed data structure for intra- and inter-process notification of label changes has accomplished this with only a few percent time overhead.

Space overhead is another matter. A production-size kernel is considerably bigger than before: about 16K of extra text and nearly 400K extra data for a 500-process system. In partial compensation, uareas are smaller. To accommodate labels, inodes on disk have been doubled to 128 bytes.

In effect labels flow along with data. Upon *exec* a process begins with the lowest label possible: the least label that dominates both that of the executed file and that of the arguments. The arguments, of course, have the label of the parent process. However, if no arguments are supplied, as for an ordinary filter, the argument label is taken to be the minimum, or bottom element of the lattice. Thereafter the label of a process changes to keep up with the data that it reads.

---

which produces binary code in the last character of the file names:

```
dev    kbytes used   free   %
disk0  5044   4124   920    82%
disk1  4984   4420   564    89%
disk1  4984   4420   564    89%
```

or, much more quickly, in the clear on the standard error:

```
df secret news 2>unclassified

dev    kbytes used   free   %use
cannot open /dev/secret
cannot open /dev/news
```

(Notice that the *open* system call does not read; *stat* does.) In particular labels may propagate through pipes.

Similarly files are created with the bottom label. (We accept a narrow covert channel through the mode field.) However, the label of the directory in which a new file's name is recorded must dominate that of the creator; the name could bear secrets.

Covert channels

Having classified many communication paths as "covert channels," we have an obligation to recognize generic classes of covert channels and to characterize their effectiveness. This we have done. Aside from very narrow "timing channels," most of the covert channels in our system involve unusual behavior: forking enormous numbers of processes or opening enormous numbers of files[5]. Thus any extensive use of covert channels should be detectable from audit records.

A mole could certainly use such covert channels to smuggle out precious small secrets to unauthorized users; however an unauthorized user could not exploit them unaided, except by planting a Trojan horse. We supply a special featureless shell to holders of the most powerful licenses to help keep them away from horses.[6] We have also designed audit tools along familiar lines to monitor the stability and safety of security settings.

We undertook this project because we believe it is desirable to try other models than those implied by thoroughgoing adherence to the Orange Book. In particular we suspect that a faithful Orange-Book UNIX system would sacrifice much of the system's productive flavor, with security barriers surprising users at every turn. Dynamic labels should help alleviate the surprises. Moreover, faithful Orange-Book security may be inappropriate in applications where security breaches do not entail risks as final as military defeat. (Commercial users, for example, may recoup damages in court.) In such a setting security priorities are more likely to concern keeping outsiders out, preventing inadvertent leaks by insiders, limiting the chance for mistakes by superusers, frustrating attempts to plant Trojan horses, and reducing the vulnerability of the overall system to a single disaffected superuser—all while maintaining high productivity. Procrustean solutions to curtail covert channels are not so critical.

---

[5] One example: create a collection of files named *A*, *B*, *C*, ... each containing one letter, *a*, *b*, *c*, ... A high process opens files to spell out a message and does an *exec* with no arguments. The resulting low process reads from the open file descriptors to receive the message at several hundred bits per second. The channel can be throttled by refusing to reduce the label across an *exec* with too many open files.

[6] This shell has no variables, no filename expansion, no compound commands, and no search path. The only builtin commands are *cd*, *exit*, and a command to drop privileges. Other command names must begin with slash or dot.

Multilevel Windows on a Single-level Terminal

M. D. McIlroy

J. A. Reeds

AT&T Bell Laboratories

ABSTRACT

Outboard from the secure UNIX† system described in our companion paper [McIlroy and Reeds, *Multilevel Security with Fewer Fetters*] are "intelligent" terminals that contain a local operating system to support multiple windows and downloaded programs, all without benefit of memory management hardware. A program in the host mediates between (multiple) shell sessions and the terminal. To run multilevel windows, the host program needs to run as a privileged program, keep track of window labels, and monitor the trustedness of the terminal. Very small changes in the terminal program enforce mandatory security policy among windows.

Mux, the manager of *layers*, or windows, for Teletype 5620 and related terminals, poses difficult security problems. In principle, each layer on a terminal behaves as a separate virtual terminal serving its own shell and associated process group. To get the most out of the terminal, we wish to run each layer with a separate label. Hence data transfers among layers must obey the formal security policy. This is difficult because layers are mutually accessible. It is possible to copy data between layers. Worse, it is possible to download arbitrary programs into layers, and layers enjoy no hardware protection.

Mux is implemented by a pair of programs, a host part that multiplexes data transfers to the terminal and a downloaded terminal part—a multiprocess operating system in its own right. The host part multiplexes bidirectional traffic to all layers. Since it must deal with layers that have different labels, the host part must be trusted, with capability *Tnochk*. The host part deals with the process group of a layer through a pipe, which the process group sees as a terminal. The pipe obeys the same labeling discipline as would a terminal; its label is marked rigid and can be changed only by trusted processes with T_{mount} capability. To detect label changes the host process enables signal **SIGLAB**. It accepts all changes, secure in the knowledge they they must have been made by trusted processes. Each change is relayed to the appropriate layer. Upon a downward label change, the layer is reset to expunge all extant data; in effect the process group gets a new layer.

The terminal part knows only enough about labels to prevent leaks. It does not implement the full dynamic label scheme. Labels are checked on every

† UNIX is a trademark of Bell Laboratories.

attempt to cut and paste data between layers; attempts to copy downward are ignored. As an extra precaution in the face of a shared address space, the terminal part erases all storage as it becomes free, including screen bitmaps, downloaded programs, and displayed text. No logging of actions at the terminal is provided, however.

Programs to be downloaded into layers run in the native hardware of the 5620 and have access to the entire address space of the terminal. Hence code run in the presence of multiple labels must be trusted. An untrusted program may be countenanced only if its label is identical to the label of all data in the terminal; otherwise it could write down to or read down from other processes. Moreover, nothing can prevent untrusted code from modifying the terminal part of *mux* itself. Thus the terminal, which becomes untrusted as soon as untrusted code is downloaded into it, must remain untrusted forever—or until rebooted. The labels stick at the untrusted value. In practice we are more stringent and require all labels in an untrusted terminal to have the same value as the the initial label of the terminal.

To separate concerns, *mux* was designed so that downloading would be done through it, not by it. Yet, to trust the terminal, *mux* must assess the trustability of downloaded code. Thereafter downloads into layers are handled by a trusted specialist program, *32ld*, that passes data through *mux*. *Mux* ascertains the trustedness of the downloader program and its connection to *mux*. The downloader is expected in turn to determine the trustability of downloaded code. Since the main pipe between *32ld* and *mux* is shared by a shell and perhaps by other processes, we protect communications over that pipe by using the **FIOPX** (process exclusive) IO control, which prevents other processes from using the pipe until its status reverts. The downloader is deemed trustworthy if it has capability T_{mount}, which it relinquishes when downloading an untrusted file. *Mux* observes the trustedness by examining indicia of privilege that come with **FIOPX**. Unless the downloader is trusted, *mux* marks the terminal untrusted.

A legitimate trusted download ends with a special coda from *32ld* which must be received while the pipe is marked for exclusive use. If an imposter kills *32ld* in mid-download the download pipe becomes unusable: the *mux* end is marked for exclusive use, while the other end reverts to permissive use. This state prevents all IO activity, in particular, attempts by the imposter to forge a download or to reassert process-exclusive access. *Mux* detects the change in state with a failed *read* system call and deletes the layer. The terminal remains trusted.

The standard version of *mux* depends on *32ld* to download the terminal part before the host part begins. We have abandoned this arrangement, which makes it difficult to assure the host part that it is indeed talking to the correct terminal part. Instead, we let *mux* do that download itself, protecting its access to the terminal with **FIOPX**. (Process-exclusive access is retained through the whole *mux* session.) This curious division of labor, wherein downloading into the raw terminal and into layers is done by different agents, is marginally justifiable: existing programs that need to load into layers already know about *32ld*, and the protocols differ, one being in hardware and one in software.

Mux also uses the **FIOPX** IO control mechanism to provide a limited form of "trusted communications path" between user terminal and application program.

While the terminal is trusted, a trusted user process running in a layer may issue a **FIOPX** call on its pipe to *mux*. As in the special case of *32ld* sketched above, *mux* detects the new process-exclusive status of the pipe together with indicia of the privilege of the process that issued the call. It notifies the terminal part, which in turn gives the layer a distinctive visual mark. The user then knows that new data typed in that layer are not accessible by eavesdropping programs. This mechanism can be used for confidential negotiations, such as password collection, that should not pass through an untrusted terminal.

Various flaws remain.

First, the 5620 itself can be subverted by downloading—before *mux* —a terminal simulator that could receive and corrupt a later download of *mux*. One way to frustrate such a gambit would be to download a tiny program that fills memory with hard-to-compute numbers, then answers inquiries about the contents of randomly chosen locations. If it doesn't answer fast enough the memory is suspect of harboring other, unwanted, code. Another way would be to modify the hardware to provide a trusted path to the native boot program. We have taken neither precaution, counting instead on users avoiding the trap by booting the terminal afresh just before starting *mux*. Aside from its reliance on human cooperation, this method is sound: the hardware of the 5620 assures that any unintended download in the interim would give a clear indication at the screen.

Second, under *mux*, the terminal practically becomes an extension of the operating system. For genuine safety, the physical security arrangements for the host computer should be extended to every *mux* terminal, and especially to the terminal's ROM.

Third, the limited trusted path for session-authorizing negotiations outlined above, will not work with an untrusted terminal. This is only a special case of a more general question: how to perform an authorizing negotiation over any external medium, the trustedness of which has not been established, and may be unnecessary for the session that follows. Challenge boxes, which accomplish confidential negotiations in the presence of eavesdroppers, are the preferred solution.

Fourth, the use of **FIOPX** to guarantee temporary exclusive access to a layer is one-sided: it gives a trusted program a way to decide that it is engaged in a private conversation with a single user. But the user has no convenient direct way to verify this. A three-phase authentication dialogue with challenge boxes could be used to solve the problem, as follows. Once the trusted program knows that the path to the user is private it issues a challenge X to the user. The user responds with $Y = f(X,K)$, and now the program knows the user is correct: he has the right K. Then, using Y as a challenge to itself, the program responds with a countersign $Z = f(Y,K)$. This last lets the user know that the program knows the secret key K of the user's challenge box, and hence is trusted, and hence the path to the program is private.

New Ideas in Discretionary Access Control

Mark E. Carson, Wen-Der Jiang

IBM Corporation

ABSTRACT

To many, it might appear that there is little else to say about Discretionary Access Control (DAC) after discussing UNIX-style protection bits and access control lists (ACL's). However, there do exist new possibilities for the use and expansion of the UNIX "setuid" concept. Presented here are two proposals.

1. Two-level ACL's

Problem Statement

The setuid bit in UNIX allows a protected subsystem to restrict users in manipulating files using server provided functions only. As a trivial example, in order to update their own passwords, users can't use *vi*, but can execute the setuid-to-root program *passwd* to edit the file */etc/passwd* in a controlled way. The problem with setuid is that in using it to restrict which programs can be used, you have to give up some of the other restrictions you might want to enforce. As one example, suppose you want to force people to use the SCCS tools to update SCCS files. To this end, you make the SCCS files owned and writable only by pseudo-user and group "sccs," and make the tools setuid and setgid to sccs. Now say you want to further restrict editing of kernel source files to the "kernel" group, but still leave them publicly readable. How can you do it? The only real way in conventional UNIX **without writing new programs** is to make copies of the SCCS tools executable only by group kernel, and probably setuid to some new pseudo-user, say "kernsccs." If you have several such group restrictions to make, things rapidly get out of hand. (See Figure 1)

Our Solution

ACL's are supposed to provide more flexible means of access control. However, conventional ACL's that are just extensions of the UNIX protection bits don't help any in this situation. The problem is that both UNIX protection bits and conventional ACL's determine access based only on the effective id's. The solution is to create ACL's which can arbitrate access based on *both* the effective *and* real id's. In the example above, we can mark the kernel source files as writable only by effective id sccs and real gid kernel. (See Figure 2) (In Berkeley-style systems with group access lists, the effective id is tacked onto the group list for the effective check, and the real id for the real check.) With this sort of "two-level" ACL, it is easy to put together restricted subsystems without any changes to code (except for the initial setup, which can be done by hand). It provides an extra degree of discretion for discretionary access control.

2. Setuid Devices

Initial Thought

Every inode in the file system has space for the setuid bit. However, this bit is interpreted for binary executable files only. Could the setuid bit be used for other things as well? Many would like to have setuid shell scripts, but in conventional UNIX nobody would want to pay the price of giving the shell such privilege. (In Secure XENIX, where one can isolate the privilege to perform *setuid()*, this is a much more reasonable idea, though it is not implemented.) Another new idea in this area is the establishment of "setuid objects."

Basic Idea

The basic idea is that a setuid object, instead of changing your id to match its, changes its id to match yours. In other words, when you first open a setuid object, the kernel gives its ownership and access to you. (The ownership change is made only in memory — in the incore inode — with the old ownership information stored in an otherwise unused area.) You retain ownership and access rights. While you "own" it, you can do anything you want with it, including changing the access rights, giving it away, etc.

One Application

The main applicability we see for this idea is devices. Many devices, such as communication devices, should be publicly accessible, but once employed by a particular user should be reserved exclusively for that user until relinquished. The usual way to handle this situation currently is by some sort of locking mechanism implemented either at the application level or in the device driver. The problem is that locking lacks flexibility; at best, it's really a sort of crude device-specific concurrency control mechanism. In particular, it doesn't allow any possibility of simultaneous sharing of a device between two users. With setuid devices, the system access control mechanisms can be used both before and after opening to control access. This can include methods of arbitrary complexity, such as ACL's. No changes to applications or device drivers are needed to take advantage of this concept. In fact, it can be installed by simply relinking the kernel, even on ordinary XENIX systems. And since the change takes place only in memory, no cleanup is required after system crashes.

The setuid device concept is sufficiently sophisticated that it can be used to handle such chores as permissions on a terminal before, during, and after login, reducing the need for privilege in *init*, *getty*, and *login*, and eliminating some holes in the current UNIX implementation.

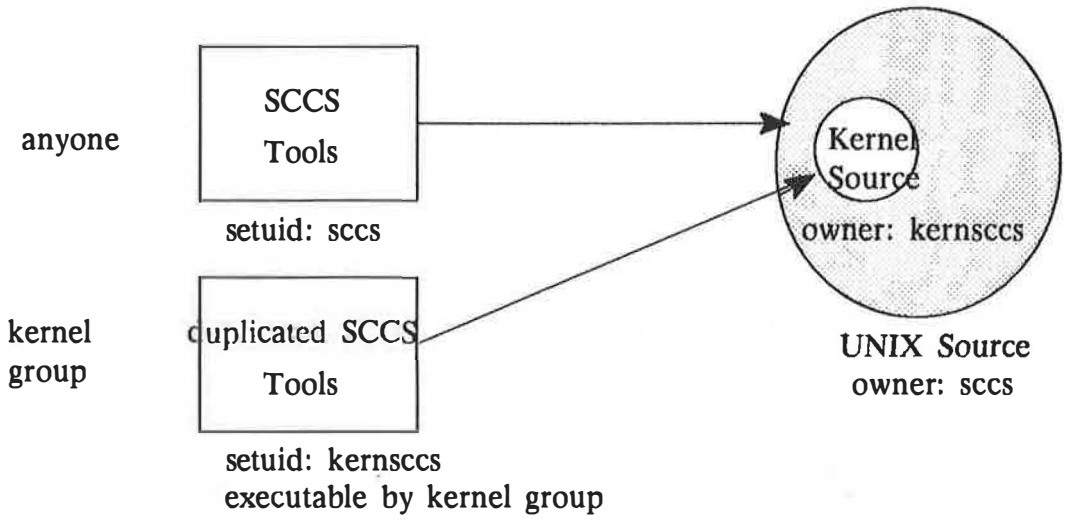

Figure 1.

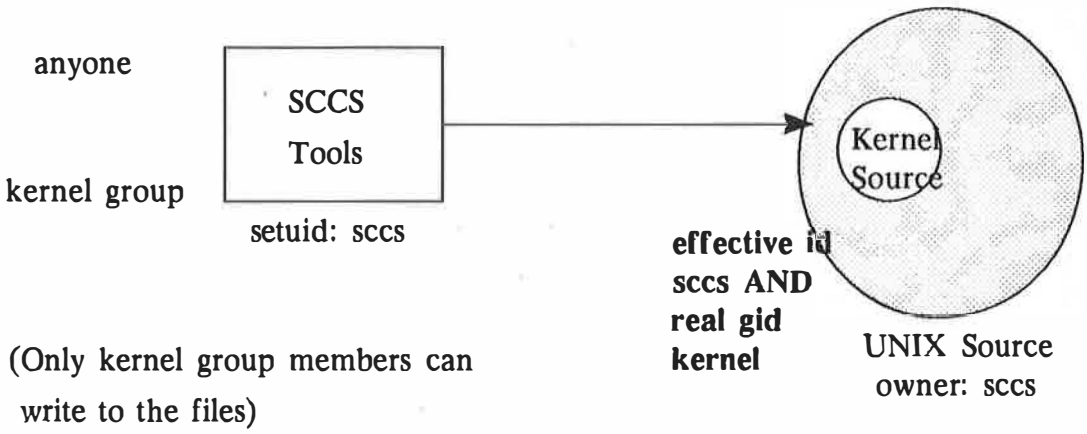

Figure 2.

On Incorporating Access Control Lists into the UNIX Operating System

Steven M. Kramer

SECUREWARE, Inc.

ABSTRACT

UNIX vendors are currently discussing ways to place Access Control Lists (ACLs) into the UNIX Operating System. Various observations and proposals have been put forth, yet none has been adopted as a standard or basis for a standard. This paper does not propose another standard — rather, it examines the options that are available in considering both the current form of discretionary access control in UNIX and the effect of including ACLs in secure UNIX.

1. Introduction

Many UNIX vendors are becoming interested in securing the UNIX Operating System according to criteria in the *DoD Trusted Computing Security Evaluation Criteria*, also known as the *Orange Book*. Vendors are primarily developing systems targeted for the C2 and B1 classes, with some ambitious vendors considering efforts at the higher classes of the B division.[1] Although access control lists (ACLs) are not required until the B3 class, many vendors are interested in ACLs as solutions to UNIX discretionary control requirements at lower classes. For this reason there is great interest in forming a UNIX ACL standard rather than having multiple and possibly incompatible ACL models and implementations.

While the secure UNIX standardization task set up by such bodies as the IEEE and /usr/group are commendable, it is this author's position that the efforts are proceeding too quickly. Because ACLs are largely a new, untried concept in production UNIX operating systems, and because UNIX uses a simple form of discretionary access control, the UNIX community should first open a forum on the effects of the placement of ACLs in the UNIX system on the users, administrators, existing UNIX security policy, program compatibility and ACL mechanism portability. Solutions are being proposed as ends in themselves rather than as solutions for a common framework adopted by the vendors. The presentation of solutions, while provoking discussion, serve to obscure the real questions that must be faced prior to accepting interfaces that achieve the intended purpose.

1. The *Orange Book* rates systems according to four divisions called A, B, C and D, with A the division requiring the most features and assurances. Within each division are classes: C1, C2, B1, B2, B3 and A1, with C2 and B3 the classes with the most requirements within their respective divisions.

The danger in the adoption of a mechanism or interface before careful consideration is that the mechanism or interface will be found to be defective in practice. Vendors will through natural selection find their own solutions that will ultimately defeat the purpose of the current efforts. The difference between the ACL standardization effort and that of the IEEE P1003.1 committee is that the latter has worked from experience with many implementations, with some implementations in existence for 20 years, and that there is a desire among users and vendors to converge on some central aspects of the operating system. In the ACL standardization effort, there is limited experience in using the mechanisms in other than research, or classified (and thus silent) sites, and in proprietary systems not subject to the same acceptance as UNIX.

It appears that ACL solutions are being considered without first defining the goals. This paper will attempt to introduce the options that are available and suggest that discussions on an ACL interface proceed at this level before solutions are presented. The next section defines the mechanisms that are key to the analysis. Section 3 enumerates the options that are available in combining the mechanisms. Section 4 presents some early observations that serve to direct future discussions. Finally, a summary of the progress to date is provided in section 5.

2. The Mechanisms

This paper is concerned with *discretionary access control*, the mechanisms of a system under which a user has control over access to his resources. Discretionary access control is contrasted with *mandatory access control*, where the policies for access control are determined and enforced by the system and cannot be overridden at the discretion of a user.

In traditional UNIX, discretionary access control is achieved by providing access by the owner/group/others model. Each file and IPC object has permissions that may be assigned to the owner of the object, to users in the same group as the object, and finally to everyone else. The permissions allowed are generally read, write and execute, with some minor variations on some of the IPC permissions. For the purposes of this paper, let us denote this model as the OGO model. There is no mandatory access control in traditional UNIX.

Access control under OGO may specify only one user and group, and the user is limited to being the owner of the object. Furthermore, the order of access checking is fixed to first check the owner of the object, then the group, and finally all others. In the OGO model, the owner of a resource could not specify access with respect to an arbitrary user. The group mechanism, of course, may be set up to form user associations that can alleviate some of the problems with the coarse granularity of the OGO model, but there are some problems with using groups as a crutch for the long term solution to the problem:

1. In order to allow or disallow access to an arbitrary user, one would need an **/etc/group** file constructed with enough groups to be the power set of the number of users. If there are n users on the system, one would need

   groups to handle all possibilities.

Even allowing for a full 16 bits for the number of distinct Group IDs, the maximum number of users allowed on the system would only be

2. For security reasons, the group file is maintained by administrators. Users do not have direct control in forming user associations. Given that all potential groups are not found in **/etc/group**, the owner of an object needs to over restrict or under restrict the object, or make an administrator request each time a new group needs to be made.

3. As users are added and removed from the system, or as they migrate to different groups, the protection a user places on an object may become out of date with the changing circumstances. There is no easy way to recalibrate the object permissions with the changing group composition. Some files may need to follow the group regardless of the members, while others are tied implicitly to the group membership. There is no record in UNIX that lets one distinguish between the two cases.

4. There is a naming problem with large amounts of groups; it is difficult to remember the membership requirements for a group given the name itself.

5. Subroutine support for the OGO model will not handle the extra burden of placed on the group mechanism. For example, the *getgrent* (3) routines used a fixed size internal buffer to hold the ASCII login names for users in a group. The fixed size is not sufficient for the potential number of members in a large group. Also, a large number of group entries would make the linear file scan of the *getgrent* routines quite slow.

ACLs provide a better way to exclude or include access to the granularity of an arbitrary user as required by the *Orange Book*. It also alleviates or eliminates some of the problems encountered in using the group mechanism for many cross purposes. An ACL is a list of specific

(*user* , *permission*)

or

(*user*, *group*, *permission*)

entries. The second form allows access not only to the individual user, but additionally to a user/group combination. The second form of ACL is the one mainly of interest to UNIX vendors.[2] Implementations of ACLs permit a wildcard in the *user* or *group* field as a shorthand representation for denoting all members of a group or the user account with all groups, respectively. The *permission* field must be able to include or exclude any of the permissions in the model.

---

2. **The first form has some of the problems related to the groups of the OGO model. Users must be listed explicitly, which is tedious on large systems. The second form allows identification according to specific users or groups; the owner of the object can signify via the ACL if it is the user or group that is more important in the discretionary label.**

An ACL model must specify how to order the entries in the list and specify what combinations of ACL entries are legal. In an ACL model with ordering, the ACL:

(smk, sware, NONE)
(smk, sware, ALL)

is interpreted quite differently if the list is read *as soon as* a user/group match is found or until the *last* such match is encountered. On the other hand, if the ACL model places no ordering requirements on the list, the model cannot allow such ambiguities in an ACL, since nondeterminism is not consistent with security goal of a clear and simple determination of access.

3. The Options

The major problem in the inclusion of ACLs into the UNIX discretionary model is that the result must reflect some combination of two distinct discretionary models. Placing ACLs into a new operating system is non-trivial because there is no compatibility struggle with existing discretionary controls. This section lists the categories under which one may combine ACLs into UNIX. Prior to an ACL solution on UNIX, one must choose a category in this section. The categories may be broadly divided into treating the discretionary policies as *independent* or as *derivations*:

3.1 Independent Discretionary Policies

These categories treat the OGO and ACL models as separate policies that are managed independently. However, because both determine access rights, there is interplay in the way the models work. Some of the methods of integration are far more plausible than others, but several approaches are included for completeness.

- One possibility is for the discretionary policy to handle OGO only.

  This is the case of traditional UNIX. The problems with going solely with this approach is detailed in the group discussion of the previous section.

- The converse is for the discretionary policy to handle ACLs only.

  This is the cleanest model theoretically because its discretionary control is based on a single, powerful model. However, it presents other problems. The compatibility with traditional UNIX is lost. All programs that rely on the mode word of a file or IPC entity will produce erroneous results. Because the number of programs (some highly trusted) that depend on the OGO form of access is very large, it would be difficult to transition to another model. Furthermore, the ongoing industry efforts in standardization of the base operating system do not seem to lean towards the elimination of the OGO model.

- Both the OGO and ACLs are present in the system, but only ACLs are used in access decisions.

  This is basically the case above, without completely eliminating the OGO model

from the system. The difference in the two methods is that this method has compatibility at the representational level with the OGO model. However, the OGO model is completely ignored in making access decisions, and as such, is not really part of the discretionary mechanism. The "compatibility" provided here can be a danger for programs that must actually make use of the OGO user, group and mode words on objects, because the values there cannot be believed.

- Both the OGO and ACLs are present in the system, but only the OGO model is used in access decisions.

  This may be a great way to transition to a composite model approach in a development or experiment, but provides no extra functionality from traditional UNIX. Since the ACLs are non-functional in the discretionary policy, they only serve to complicate the mechanisms. The system does not allow restricting access decisions to the granularity of an individual user.

- The discretionary policy is the conjunction of the results of both the OGO and ACL access checks.

  Only when both methods allow access would access truly be granted. The benefit of this approach is that both compatibility and security issues are addressed here. The disadvantages are that neither model reflects the actual protection on the object. Existing programs that believe the mode word of a file to determine its accessibility may mistakenly assume that access is allowed when it is really denied.

  The disadvantages magnify an otherwise small security problem for traditional SUID programs. A SUID program that has full access to an object may modulate access to others through interpretation of the user, group and mode words of the object. If the SUID program now were not to consider the ACL on the object, access may be granted prematurely. Having a SUID program, say acting as a trusted program, perform such discretionary access checking is a case where the implementation of the model is decentralized. By changing the underlying discretionary model, vestiges of the original decentralization may not operate correctly in light of a new, composite model.

- The discretionary policy is the disjunction of the results of both the OGO and ACL access checks.

  If either model allows access, access is allowed overall. The advantage to this method is that no matter what model one uses in determining access, an affirmative access decision by the model may be believed. The problems in the previous case with the SUID program are not an issue here. The disadvantage is that when determining the converse, that is, who is *denied* access to an object, this method is not accurate when adhering to only one of the models in the composite. Because security policies should always err on the side of caution, it is extremely dangerous to for a user to explicitly deny access of an object under one model only to be forced to remember to do the same with another model. One may expect many serious omissions resulting from this method.

- An object may carry either OGO or ACL markings, but not both.

  This is the approach used by secure XENIX [2]. The advantage is that, on a per-object basis, the discretionary policy is clear. Exactly one of the models applies to the object, without interference from the other model. The disadvantage is that there is not one global policy governing all system objects. Design problems such as the interpretation of a mode word during a *stat* (2) call for an ACL-only file must be solved.

3.2 Derived Discretionary Policies

These categories show the result when one of the OGO or ACL models is derived from information in the other model. In these cases, only one of the two models is independent. The other one is a transformation of the data and policies of the independent one.

- The ACL is the independent model and the OGO uses the least possible accessibility, or covering, without causing a violation of the ACL access policy.

  The OGO owner, group and mode word are derived from the ACL each time the ACL is modified. The actual modes in the mode word are such that the greatest permission is allowed that would not permit access otherwise denied from the ACL. As an example, an ACL (assumed to be ordered and stops the search the first time a match is found) of a file owned by *smk*,

  (smk, sware, rwx)
  (smk, *, rw)
  (*, admin, rx)
  (*, *, r)

  would yield OGO discretionary controls as follows:

  Owner: smk
  Group: sware
  Mode: rw-r--r--

  Note that each set of three permission bits in the mode word were formed by the intersection of the permissions derived from all possible ACLs that fall into the same identity category. For instance, even though the (*smk*, *sware*) combination contains rwx access, the owner bits are only rw- because the ACL entry (*smk*, *) also maps to the owner category and it only has rw- permission.

  The only problem in this method is the handling of changes to the derived quantity. If, for example, the owner issues the *chmod* (2) system call on the object, the question arises on what to do to the ACL, if anything. The *chmod* call could be disallowed if the mode word is treated as a strictly derived quantity. For compatibility with the *chmod* function in standardization efforts, that approach may be too drastic. An alternate method is to reflect the change in the ACL, either by replacing the ACL with the exact intent of the new information in the OGO model or by supplementing information already in the ACL with enough entries to maintain the derivation maxim.

An advantage of this approach is that emphasis is placed on the more refined ACL model while still retaining semantics of the OGO model for compatibility. A disadvantage is that the OGO model is only an approximation of the true access to the system. As the ACLs are more complex, the derived mode words tend to approach ———, or no access, even if many types of access are allowed.

- A similar approach is to keep the ACL as the independent quantity but to make the derived OGO quantity cover all affirmative ACL accesses.

  Using the same example as above, the ACL

  (smk, sware, rwx)
  (smk, *, rw)
  (*, admin, rx)
  (*, *, r)

  would yield OGO discretionary controls as follows:

  Owner: smk
  Group: sware
  Mode: rwxrwxr-x

  Any access granted by the ACL is granted by the derived OGO representation. With this approach it does not take many ACL entries to have the derived mode word approach rwxrwxrwx.

  An advantage here is that if the OGO model denies access, no further search through the ACL is necessary. A major disadvantage is that an ACL as conservative as

  (smk, sware, rwx)
  (ruth, dea, rwx)
  (*, *, NONE)

  produces a derivation as liberal as rwxrwxrwx.

- The OGO is the independent model and the ACL represents this access directly as an ACL.

  For example, the OGO entry of:

  Owner: smk
  Group: sware
  Mode: rwxr-xr—

  would appear as a three entry ACL:

  (smk, *, rwx)
  (*, sware, rx)
  (*, *, r)

  This approach, while workable, puts such severe limitations on the types of ACLs that may be formed that no benefit is derived from the ACL model above the OGO model.

One possible variation of this method is that after the OGO information is set and the ACL derived, a later ACL operation can fine tune access. However, because it promotes independence between the OGO and ACL models, this expanded case is handled by the independent options.

- The ACL is independent and the OGO data is derived from entries that match an OGO-like pattern.

  The patterns would be ACL entries of the variety:

  (*owner*, *, *permissions*) ,
  (*, *group*, *permissions*) ,
  or
  (*, *, *permissions*) .

  Sensible algorithm choices are form the OGO permissions from entries that either:

  — First match one of the patterns above.

  — Has *more* permissions than any other ACL entry of the same kind.

  — Has *less* permissions than any other ACL entry of the same kind.

  Variations on this theme could also work.

- The ACL is the independent model, and the OGO effective permissions are read from the point-of-view of the invoker.

  Processes that issue the *stat* call on a file or issue an **IPC_GET** command on an IPC entity will get back data in the traditional form. However, that data will be derived from the user's perspective. For instance, the file owned by (smk, sware) has an ACL of:

  (smk, sware, **rwx**)
  (smk, *, **rx**)
  (hal, *, **x**)
  (*, sware, **rx**)
  (maintsup, *, **rw**)
  (*, maint, **rw**)
  (*, engr, **x**)
  (*, *, NONE)

  when the *stat* call is invoked by (smk, sware) would yield:

  Owner: smk
  Group: sware
  Mode: **rwxr-x---**

  For (hal, sware), the same file produces:

Owner: smk (or any other user besides hal)
Group: sware
Mode: —r-x--

and for (fred, engr) would yield

Owner: smk (or any other user besides fred)
Group: engr
Mode: ——x—

The real permissions as applied to this behavior may be the same as the effective permissions, or they may more accurately map the "system view" of the ACLs to the OGO, where the owner, group and mode of the object are true to the ACL. The second alternative can be beneficial to SUID programs that need to obtain accurate labels on both the subject and objects in order to make access decisions.

4. Observations

The observations in this section relate to the options in the previous section, the denial properties of ACLs, and the extensions of permission types. The latter two topics are worth consideration because the ACL is a new mechanism that does not have the compatibility concerns of existing UNIX mechanisms, and while they may not have analogues, should be examined in this light before the process of standardization is complete.

4.1 Plausible Options

From the options presented in Section 3, one can admit that there is no one solution that best addresses compatibility with traditional UNIX, a wide range of functionality, and increased security. These goals must be prioritized before the selection process is narrowed. If compatibility can be sacrificed, the ACL mechanism by itself is a simple model that satisfies the *Orange Book* requirements. However, most vendors have an established user base that cannot be ignored. The question then becomes how much compatibility can be traded for a new composite model? It is clear that with any of the realistic composite models discussed, there are degrees of compatibility, but none of them offer complete backwards compatibility. Some of them are also more usable than others.

In light of there being no one approach that presents itself well over the others, perhaps the answer is to allow multiple approaches. This can be handled in a number of ways. A system could be advertised with a particular approach or may support multiple approaches at compile time, at boot time, or dynamically on a running system. The intrusions of competing discretionary policies is a big factor in combining approaches. Any of these is a possibility, but much work would need to be done in providing a platform for portable programs that would have to recognize several policies. Such alternatives to the single policy approaches have not been explored.

4.2 Denial Properties of ACLs

Even with the many possible mappings between ACLs and the owner/group/others models with respect to the *allowance* of access, ACL models that are deemed useful allow for the access *denial* of permissions for (*user*, *group*) pairs. This is an attribute in ACL models that may make the mappings and composite policies of Section 3 more complex.

4.3 Permissions

The options in the previous section assume that ACLs will contain the same read/write/execute permissions as in the OGO model. Because the ACL model is new to UNIX, there is no reason to restrict the permissions to that of the restrictive OGO model. Designers of an ACL model must look into new permissions such as these:

Ownership
: The attributes of ownership can be spread among several users, allowing each to perform the extra services on objects that a single owner (or the superuser) can now perform, in addition to setting this and other permissions described below:

Extend
: This permission would be needed to make objects grow in size. This is useful especially when one user opens up files to other users, but is concerned that the other users may consume resources that will be charged to that user.

Truncate
: A user would need this permission to shrink a file. A user that could write to an object could not delete (or overwrite) existing information. Such a use would be for log files allowing untrusted users to write them but not allowing previous information to be lost.

Delete
: The file can be removed only if the user contained this privilege. Currently, there is no "delete" permission for files; deletion is accomplished by being able to write into the directory containing the file. The delete permission would be checked in addition to the directory permissions before the file is removed. The delete permission for IPC entities would be needed before the **IPC_RMID** command could be used for removal, and for processes it would be needed before signals could be sent to the process.

Delete By Owner
: Only the owner could delete a file. This permission would appear in the directory containing those files. Some systems implement this feature now by using the "sticky bit" on directories; this permission would make such use explicit.

OGO
: An ACL entry containing this permission would have to be of the form:

(*owner*, *, *permissions*+OGO)
(*, *group*, *permissions*+OGO)
(*, *, *permissions*+OGO)

and exactly one of each type would be present in the ACL. These three forms would constitute the user-determined OGO derivation. Unlike the cases in Section 3 that promote dependence through fixed derivation, this plan would allow the user to describe the derivation explicitly. Some of the problems of Section 3 remain, such as how the policies combine to form the composite model. Also, in the absence of entries with the OGO privilege, there must be a deterministic policy — the choices include all of the options of Section 3.2.

Some of the permissions are truly new to UNIX; others are merely clearer representations of permissions that were previously overloaded into the rwx scheme. By making such permissions overt, UNIX users could see the permissions for exactly what they are rather than having to master a mapping to the rwx mechanism for some esoteric privileges.

5. Summary

This paper has presented the major options possible in forming a composite UNIX discretionary policy from the unique properties of both the OGO and ACL models. While no options have been eliminated, it is hoped that some options can be eliminated quickly as being unusable, leaving a few options for serious discussion.

It must be argued that, only when the approach or approaches are agreed upon for standardization, can solutions to the problems be examined and adopted. It is much more dangerous in the long run to present actual solutions as the standard and force the observer to infer the model under which it works — a solution may satisfy multiple models or contain complexities not readily apparent.

Finally, the incorporation of ACLs into UNIX should not stop at allowing/disallowing access at the granularity of a user and also provide compatibility. If there are other concepts that would make ACLs more usable, they should be included before standardization stops further diversity. This paper has mentioned some aspects dealing with permissions that are important topics that have not been central to the discussions on standardization up to this point. Sensing that ACLs will be a central component of the UNIX Operating System for at least the next 20 years in its life, it is advisable to invest more consideration by the community prior to choosing a standard.

Miro: A Visual Language for Specifying Security

C. A. Heydon, M. W. Maimone, A. F. Moorman,
J. D. Tygar, J. M. Wing

Computer Science Department
Carnegie-Mellon University

ABSTRACT

Miro is on ongoing project at CMU to design and implement a visual specification tool for security constraints. The visual format provides a mathematically precise notation for expressing security constraints which can be easily understood and modified by users who are not specially trained in security formalisms. The language contains a typing system which allows a system administrator to restrict all potential security configurations to follow a specific pattern, including standard mandatory security requirements.

One important feature of a visual language is the straightforward representation of hierarchical issues. Miro uses diagrams as a natural way of showing security relationships between users and data; for example, Miro allows a user to separate the security at a large site into visually distinct subparts. This allows complicated security structures to be presented at various levels of detail.

Miro can be used to reflect any sort of security structure that can be expressed as a set of contraints on an access control matrix. It is not specific to any system. Miro can also be used to reflect the dynamic nature of security; the system allows one to express the accumulation, deletion, and modification of security structures.

Our talk will give an overview of the language and semantics, and a discussion of the tools we are implementing to support the use of our language in real environments.

StrongBox: Support for Self-Securing Programs

B. S. Yee, J. D. Tygar, A. Z. Spector

Computer Science Dept.
Carnegie-Mellon University

ABSTRACT

Security is a pressing problem for distributed systems. Distributed systems exchange data between a variety of users over a variety of sites which may be geographically separate. A user who stores important data on processor A must trust not just processor A but also the processors B, C, D, ... with which A communicates, and the various media M_{ab}, M_{bc}, ... by which these processors communicate. The distributed security problem is difficult, and few major distributed systems attempt to address it. In fact, conventional approaches to computer security are so complex that they actually discourage designers from trying to build a secure distributed system. A software engineer who wishes to build a secure distributed data application finds that he must depend on the security of a distributed database which depends on the security of a distributed file system which depends on the security of a distributed system kernel, etc. It is hard just to make a distributed system work efficiently without considering security issues.

We have constructed and are using trusted application systems that run efficiently on machines having only minimal security facilities. Rather than depending on a tight kernel of security code, our applications perform operations that allow us to guarantee security. We call these trusted programs *self-securing*. By limiting dependence on underlying system components, we separate the security problem from the rest of the system, simplify the task of the engineer who must build such a system, and allow existing distributed systems to be retro-fitted with security. Our concern here is with security issues arising from protecting the privacy of data and the integrity of data from alteration. We do not presently consider issues of denial of service, covert channel analysis, or traffic analysis of message patterns, although we are extending our work in these directions. An important assumption in this work is the integrity and privacy of address spaces.

We have developed a family of algorithms that support self-securing programs. To show the effectiveness and efficiency of our methods, we have implemented them in a package called *Strongbox* and measured their performance in our computational environment. We have successfully built on two ongoing systems research projects at Carnegie Mellon University: Mach, a distributed operating system which is upward compatible with 4.3 BSD UNIX; and Camelot, a distributed transaction facility which runs on Mach. We do not assume that our base operating system provides full security. Our tools allow users to run application programs securely on Mach and Camelot. To demonstrate our tools, we have built a full protection system for Camelot which runs with negligible overhead. This protection system has been distributed with Camelot.

Auditing Files on a Network of UNIX Machines

Matthew A. Bishop

Department of Mathematics and Computer Science
Dartmouth College

ABSTRACT

The Numerical Aerodynamic Simulator project runs a variety of UNIX† based operating system on its computers (a Cray 2, 2 Amdahl 5840s, 4 VAX-11/780s, and 25 IRIS 3500 workstations, all connected by a local area network and connected to a number of wide area networks such as ARPAnet, BARRnet, and various others. Within this environment, much development is done on each machine, particularly by engineers who come from outside Ames. They are not always aware of (or respectful towards) the policies of computer security the NAS Project has set up. Worse, given the networks to which Ames is connected, an attacker who could subvert the network controls and break security could leave traces in the form of altering files in system areas (for example, to make gaining access to the system a second time easier.) For these reasons, we decided to establish a file tree auditing system.

The audit system works as follows. It scans a file system, listing name, type, protection mode, number of (hard) links, user, group, and time of last modification. The results are saved in a file, and this file is then compared to a file with the same format but containing a snapshot of expected results. Any differences are mailed to the appropriate people; they must take action to determine what to do.

The audit system is stored in its own subtree and contains several files and subdirectories. The file *Environ* contains the location of the programs the auditor uses (namely, *lstat* , which generated the listing for each file; *auditls* , which collates the listings for the file system; *egrep* (1), *ls* (1), *find* (1), and *test* (1), the UNIX system utilities.) The file *List* lists the roots of the file trees to be audited, and for each specify a set of options to the audit program; these options are applied only to that file tree. Master files reside here too, and are named by deleting all "/" characters from the name of the root of the file tree, and prefixing the letter "F". (If only setuid files are to be audited, the prefix is "FU"; if only setgid files are to be audited, the prefix is "FG"; and if both types of files are to be audited, the prefix is "FB".) Also here are ignore files; these files are named the same as the corresponding master files (but the "F" is replaced by an "I".) These files contain regular expressions that are used to eliminate uninteresting files.

When we had a system with which we were satisfied running on one machine, we expanded it to run multi-machine audits. This required reorganizing the program and the file structure in the audit subtree. We decided to run the audits in a master-slave relationship; the master would issue a command to the remote host to

Work reported here was supported by NASA under contract NCC 2-398 and was done at RIACS, NASA Ames Research Center, Moffett Field, CA 94035.

† UNIX is a trademark of Bell Laboratories.

execute a program (actually, its version of *auditls*) and send the output to the requester. This required two programs, *auditls* and *lstat*, to be available on the remote host, so we updated the installation procedure to do this. We also had to define the mechanism to execute commands remotely; since the System V based machines used a different command than the 4.3 BSD based machines, we made this an installation time parameter. We also put the *Environ*, *List*, master, and ignore files for each machine into a separate directory, and created an *Equiv* file to map host names to one another, so (for example) the same machine could be referred to as *icarus* or *icarus.riacs.edu*.

We quickly discovered two problems with running audits remotely. Both came about because some portions of the network software being developed were unreliable. Either the network would hang, leaving the connection alive and hung, or the network connection would be broken before the results of the remote file system scan had been completed. In the first case, the auditing process would be stopped dead in its tracks; in the second, a very large number of files would show up as being deleted, and then show up again the next day as having been created!

We dealt with both problems by making allowances for them in software. For the first, we wrote a timeout routine that executes a command, waits for a user-specified time, and then (if the process is still active) kills it and reports the termination. There is a danger that this might prematurely terminate remote file system scans running on slow or heavily loaded machines; but the timeout was set to 1 hour, and that proved to be sufficient to kill only hung processes. For the second, we made the assumption that the file systems and directories being audited changed in small increments only. So, we added a threshhold parameter which took action if the number of files in the remote file system were under a certain percentage of the number of files supposed to be there. For example, if the auditing system reported that directory */bin* on machine *chewy* had 60% of the files it was supposed to have, the results of the file system scan would be saved somewhere, and a message put in the results of the audit. The message reads: "There is a potential problem with the file system /bin on chewy — the audit showed that file system has 60% of the files it had when the master was made. Either the audit failed or most files on that file system have been deleted. Check to be sure it is not the latter, and if the master file must be regenerated, delete the current one and replace it with results.bin. Note: the master files have not been updated."

Current experience proclaims this system a success. Since the addition of the features handling the two problems described above, there have been no errors in the file audits that have not been flagged as potential errors. It has caught numerous cases where developers made private copies of privileged programs and disabled their security features. The system has been in use for about a year, and has paid off handsomely.

An Experimental Trusted Path Prototype

Brian Foster

The Santa Cruz Operation, Ltd., London

1. Introduction.

A modified version of System V shell layers could be used to provide a trusted communications path. A shell layers-based trusted path requires no additional hardware, such as an additional key or a bit-mapped display, and can be used over dialup lines. Only a small number of modules need to be modified, and the resultant trusted path is very flexible. For instance, supporting simultaneous sessions at difference sensitivities should be reasonably simple. A "fast prototype" has been built to demonstrate the concept.

2. Problem.

2.1 Motivation.

To do some operations, the user may wish to be (or must be) in direct contact with the TCB; i.e., communicating with trusted components of the system over a channel that cannot be spoofed, intercepted, or blocked by unauthorized subjects. Two such events are logging onto the system and changing one's password. In both cases, a very simple program masquerading as *getty* (plus *login*) or *passwd* can easily steal the user's password. If, however, the user were to first ensure that program asking for the password is indeed the trusted program, and ensure that only that program will read the password, such attacks should fail.

A trusted path provides such a guarantee. The trusted path can be established any time the user chooses. Once established, all communication with the user is tamper-proof and occurs only between the user and known trusted subjects (such as the system's real *passwd* program).

2.2 Goals.

A reasonable trusted path should:

- Be simple to implement, and not require any "special" hardware. In particular, it should work with a teletype over a dialup modem connection.
- Provide — on demand — both a trustworthy login procedure and a trustworthy version of other operations which are exceptionally risky if not done over a trusted path.

- Should have low overhead and little performance impact.

A fast prototype of a system that appears capable of meeting these goals was built by modifying *getty*, *login*, shell layers (both the *sxt* driver and the *shl* utility), and all tty-style communications drivers.

3. Prototype.

The existing prototype, built on a standard SCO XENIX 386 system, does not provide a true trusted path. However, the solutions to problems that prevent it from being a true trusted path appear to require notions which should exist in a trusted system, such as privileged signals and forced file closure.

3.1 Details.

Whenever the user wants a trusted path established, a "hot" key sequence is typed. In the prototype, this sequence is just one key: The shell layers *VSWTCH* character (usually Control-Z).

The *login* program works as normal except that it *exec*'s a modified version of *shl* instead of the user's shell defined in */etc/passwd*. The modified *shl* (called *tpl*) is identical to *shl* except that:

- It sets both the real tty and the controlling sxt to exclusive-use, to prevent further *open*'s, and thus prevent interception.
- It lacks both the *block* and *unblock* commands.
- Every sxt is always set to block on output if not the current layer.

In addition, a feature present in XENIX but not in all other implementations of shell layers is used: The XENIX *shl* uses the value of *$SHELL* — which is set by XENIXs *login* — instead of a hardwired path of */bin/sh* as the name of the shell to run whenever a new layer is created.

A modified version of the *sxt* driver, intended for use only with *tpl*, is used. This modified *sxt* driver disallows the changing of the *VSWTCH* character (forcing it to always be Control-Z) and forces *LOBLK* to be set.

As a result of these two sets of changes, when a user logs in, *tpl* is always run. The terminal is exclusive-use (*XCLUDE*). Hence the user is guaranteed that no process can open the line to intercept or spoof the communications. When a layer is created, it does not inherit any open file descriptor and so cannot affect the setting of *XCLUDE* or *LOBLK*. Thus, one layer cannot intercept or spoof the communications with another layer.

Each layer created could, in principle, operate at a different sensitivity. Additional commands could be added to *tpl* to perform operations which must be done while in the trusted path (such as changing one's password).

3.2 Problems.

There are several obvious difficulties with this approach:

1. The terminal could be opened by an undesirable process before the exclusive use is established.

   A primitive similar to IBMs *vhangup* or SecureWare's *stop_io*, which terminate all open connections to a specified device (so that any further *read* or *write* returns an *EBADF* error) should address this issue. While *XCLUDE* is a convenient prototyping tool, perhaps, in a real trusted path, it should not be used to enforce the necessary exclusion.

2. *tpl* could be killed.

   Preventing the delivery of signals to a more privileged process should solve this problem. In a trusted system with a fine-grained privilege scheme, *tpl* would presumably have at least a "set operating sensitivity" privilege.

3. When is the user's *.profile* run?

4. This trusted path scheme exists only after the user has logged in, but logging in is one of the times that a trusted path should (must?) be used.

   The solution implemented by the fast prototype is to modify every tty-style communications driver to provide a *T_SWTCH* device command which sends a *SIGUSR1* signal. So if the user types Control-Z while *getty* is running, *getty* is sent a *SIGUSR1*. Once *tpl* is running, the modified *sxt* driver intercepts the *T_SWTCH* command and does not pass it along to the hardware driver.

   To force the users to use the trusted path when logging in, *getty* was modified to say "Please type Control-Z to log in" and not proceed until a *SIGUSR1* is received. In order to handle the traditional cycle-baud-rates-on-BREAK function, *getty* actually reads (but does not echo) the terminal while waiting for a *SIGUSR1*; each time a BREAK is detected the baud rate is changed as appropriate and the "Please..." message is issued again.

5. The "hot" key sequence is just one character, and recognition depends on whether or not *ISIG* is set. Furthermore, because the character is recognized in the line discipline, any previous input character translation (such as European "dead" or "compose" keys) could be used to defeat recognition of the sequence. Finally, use of the real shell layers is prohibited by the overloading of the *VSWTCH* character and *T_SWTCH* device command.

   All of these problems could be solved by having each tty-style communications driver recognize a (presumably timing-dependent) multi-character sequence as soon as possible in the input stream and issuing some new *T_TPATH* command.

The prototype is not a true trusted path, mostly because not all of the necessary mechanisms exist in the standard XENIX (or UNIX) kernel.

4. Conclusion.

A simple, easy-to-build prototype of a shell layers-based pseudo-trusted path seems to offer some intriguing possibilities for building a proper trusted path. The ease with which the prototype was built and the limited experimentation that has been done, suggest that the concept is viable. Further work is necessary to formalize the *ad hoc* design and to understand the limitations present in the mechanism.

Access for Operators that Require Root Privleges (SUID & SGID)

Don Winsor

Hill AFB

The Autodin Intersite Gateway (AISG) computer system is installed at 6 Air Force sites running UTX/32 (4.2 BSD & System V) on two Gould PN6040J's. One cpu is the active gateway while the other is its backup. These systems are connected to AUTODIN (Automatic Digital Network), DDN (Defense Data Network), DCTN (Defense Commercial Telecommunications Network), ETHERNET and LAN.

The application software for the system was contracted to a company who contracted to another company. Their application software development required that the operators be super user in order to run some of the programs. The use of some commands that super user could only run were also needed in order to operate the system. Once a week the operators switch to the other cpu so that the hours on each could be approximately equal and to make sure the cpu was working. As you can imagine the problems that could develop. Before the system at Ogden was put into production I decided to minimize the risk of having the super user access know by so many. Some interesting things had happened at other Air Force bases with the same systems, someone had removed unix, unix.bak, and */etc*, someone modified */etc/rc.local*, */etc/passwd* and */etc/group*. Therefore the need for an operators menu or something was created. The commands/programs that needed to be used by the operators while in super user mode are:

bkrs	backup of application files
date	set system time/date
fastboot	fastboot system
gslget	gateway status log control
kill	kill a pid
lpc	line printer control
mirror	display mirrored disks
mount	display mounted disks
reboot	reboot system
shutdown	system shutdown
syst	system state changes

These commands/programs are executed from a menu that the operators have access to. This is how it works:

A script menu calls a compiled C program that uses Set-user-id (SUID) and Set-group-id (SGID), the C program executes a command or a script as if it was executed by root, once the command or script is done the control is passed back to the menu program. The executable C program is owned by root but is using the sticky bit in order to execute the command or script. The commands and scripts are also owned by root. The menu scripts are owned by the user.

I first developed the menu and submenu using scripts.

```
# omenu script:
cat <<!
        ISG OPERATORS MENU
    Bkrs (backup of system files) . . . .  (1)
    Date (set system time and date) . . .  (2)
    Gslget (gateway status log control) .  (3)
    Kill (kill a process id) . . . . . .   (4)
    Lpc (lpr control program) . . . . . .  (5)
    Lpq (lpr que) . . . . . .  (6)
    Mirror (display mirrored devices) . .  (7)
    Mount (display mounted devices) . . .  (8)
    Pingdctn (check dctn connections) . .  (9)
    Pingddn (check ddn connections) . . . (10)
    Ps -ax (display processes id) . . . . (11)
    Shutdown Menu . . . . . . . . . . . . (12)
    Syst (system state changes) . . . . . (13)
    Exit . . . . . .  (0)
!

echo -n "       MAKE SELECTION PLEASE ==> "
    read ans
    case $ans
    in
        1) clear;/work/opx/xbkrs;sleep 1;clear;/work/opx/omenu;;
        2) clear;/work/opx/xdate;sleep 1;clear;/work/opx/omenu;;
        3) clear;/work/opx/xgslget;sleep 1;clear;/work/opx/omenu;;
        4) clear;/work/opx/xkill;sleep 1;clear;/work/opx/omenu;;
        5) clear;/work/opx/xlpc;sleep 1;clear;/work/opx/omenu;;
        6) clear;/work/opx/xlpq;sleep 1;clear;/work/opx/omenu;;
        7) clear;/work/opx/xmirror;sleep 1;clear;/work/opx/omenu;;
        8) clear;/work/opx/xmount;sleep 1;clear;/work/opx/omenu;;
        9) clear;/work/opx/opingdctn;sleep 1;clear;/work/opx/omenu;;
        10) clear;/work/opx/opingddn;sleep 1;clear;/work/opx/omenu;;
        11) clear;ps -ax | more;sleep 3;clear;/work/opx/omenu;;
        12) clear;/work/opx/odown;sleep 1;clear;/work/opx/omenu;;
        13) clear;/work/opx/xsyst;sleep 1;clear;/work/opx/omenu;;
        0) clear;echo"Exiting Operators Menu System";sleep 1;clear;;
        *) clear;echo "INVALID SELECTION, SELECT AGAIN!!";
            echo ^G;sleep 1;clear;/work/opx/omenu;;
    esac
clear
```

```
# odown script:
cat <<!
        SHUTDOWN MENU
    WARNING THESE WILL SHUTDOWN THE SYSTEM!!
    Fastboot (fastboot system)  . . . . . (1)
    Reboot (reboot system)  . . . . . (2)
```

```
    Shutdown (shutdown system)  . . . . . (3)
    Exit . . . . . . . . . . . . . . . . . (0)
!

echo -n "       MAKE SELECTION PLEASE ==> "
    read ans
    case $ans
    in
        1) clear;/work/opx/xfastboot;sleep 5;clear;/work/opx/omenu;;
        2) clear;/work/opx/xreboot;sleep 5;clear;/work/opx/omenu;;
        3) clear;/work/opx/xshutdown;sleep 5;clear;/work/opx/omenu;;
        0) clear;echo "Exiting Shutdown Menu";sleep 2;clear;;
        *) clear;echo "INVALID SELECTION, SELECT AGAIN!!";
            echo ^G;sleep 2;clear;/work/opx/omenu;;
    esac
```

Then the C source code was developed using the setuid and setgid for only the commands/programs I wanted them to be able to run as super user.

```
# okill.c source code:
main()
{
int x=0;
int y=0;
setuid(x);
setgid(y);
system("zkill");
}
```

```
# oshutdown.c source code:
main()
{
int x=0;
int y=0;
setuid(x);
setgid(y);
system("/etc/shutdown -h now");
}
```

Additional scripts for interactive input for commands kill and date.

```
# Shell script zkill
clear
echo -n "To abort enter <cr>,"
echo ""
echo -n "Enter up to 1 pid,"
echo ""
echo -n "Enter pid then <cr>: "
echo ""
set aec = $<
if ($aec >= 200) then # pid > 200
  kill -9 $aec
else
endif
echo ""
```

```
# Shell script zdate
clear
echo -n "To abort enter 9 <cr>,"
echo ""
echo -n "Format YYMMDDHHMM, <cr>:"
echo ""
echo -n "Enter pid then <cr>: "
echo ""
set sec = $<
if ($sec == 9) then
else
date $sec # echo date
endif
echo ""
```

File permissions as set in /work/opx (operators home directory):

```
-r-x------  1 aisgopx  staff      977 May  3 07:30 odown*
-r-x------  1 aisgopx  staff     2462 May  3 09:06 omenu*
-rwsr-x---  1 root     staff    17580 May  3 09:18 xbkrs*
-rwsr-x---  1 root     staff    17580 May  3 09:18 xdate*
-rwsr-x---  1 root     staff    17584 May  3 09:18 xfastboot*
```

```
-rwsr-x---  1 root     staff       17584 May  3 09:18 xgslget*
-rwsr-x---  1 root     staff       17580 May  3 09:18 xkill*
-rwsr-x---  1 root     staff       17579 May  3 09:18 xlpc*
-rwsr-x---  1 root     staff       17582 May  3 09:19 xmirror*
-rwsr-x---  1 root     staff       17581 May  3 09:19 xmount*
-rwsr-x---  1 root     staff       17582 May  3 09:19 xreboot*
-rwsr-x---  1 root     staff       17584 May  3 09:19 xshutdown*
-rwsr-x---  1 root     staff       17580 May  3 09:19 xsyst*
-rwx------  1 root     staff         203 Apr 29 09:57 zdate*
-rwx------  1 root     staff         822 May  5 07:09 zkill*
```

chmod 4750 x*
chmod 700 z*
chmod 500 o*

Making Your Console Secure

Rick Lindsley

Tektronix

ABSTRACT

One problem with most UNIX* computers is that there is one very easy way to be root: bring the machine down to single user and there you are! For this reason (among others), physical access to a computer is usually tightly restricted. Physical security of large machines is easy: you put them behind a locked door in an air-conditioned computer room. You give out limited numbers of keys, or change the combination frequently, or provide card-key access, and you've pretty much done your duty.

But how do you provide this same level of security to desktop workstations, whose entire tape unit, disk drive, and monitor may sit on or under a person's desk? Many commonly purchased machines, such as Sun's or Microvax II's, fit this description. These machines frequently access (or are hosts of) the same sensitive data a larger, more secure machine can, but with less security. If you cannot give that person an office with a key (and many environments, whether business or educational, choose not to do that), and you cannot disable the ability to reboot single user without shutting down the machine, how can you prevent someone from simply coming in, halting your machine, and coming up single user?

If your machine comes without key protection to disable the console, you won't be able to stop someone from rebooting or even halting your machine. But with a short C program, you *can* prevent someone from bringing it up single user without providing a password.

This program has been posted to the net before already, but it is so simple and so effective it is worth posting again. One makes an entry in of

```
trap '' 2 3
ok=no
while [ $ok = no -a -f /local/chkpass ]
do
    /local/chkpass root
    case $? in
        0) ok=yes;;
        1) echo Sorry;;
        2) echo Something\'s wrong with passwd; I\'ll allow you this time.
        ok=yes ;;
        3) echo chkpass improperly invoked\; allowing root for now....
        ok=yes;;
        *) echo unknown error status from chkpass\; allowing root for now ...
        ok=yes;;
    esac
```

* UNIX is a trademark of Bell Laboratories.

```
done
trap 2 3
```

and one can secure their console from single user access without having to have source to any other system utilities.

```
#include <stdio.h>
#include <signal.h>
#include <pwd.h>

/*
 * chkpass —  prompt for a password and check that it is correct.
 *      The first argument is required, and is the user name
 *      to check against. The second is optional, and is the
 *      prompt desired. Keyboard generated signals are ignored,
 *      thus this can safely be used from shell scripts.
 */

main(argc,argv)
int argc;
char **argv;
{
    register struct passwd *pw;
    register char *pass, *encrypted;
    char *getpass(), *crypt();

    signal (SIGINT,SIG_IGN);
    signal (SIGQUIT,SIG_IGN);
    signal (SIGTSTP, SIG_IGN);
    if (argc != 2 && argc != 3) {
        fprintf(stderr,"usage: chkpass username [prompt]\n");
        exit(3);
        }
    if ((pw = getpwnam(argv[1])) == NULL) {
        fprintf(stderr,"chkpass: can't find user \"%s\" in /etc/passwd\n",
            argv[1]);
        exit(2);
        }
    if (pw->pw_passwd && strcmp(pw->pw_passwd, "")) {
        pass = getpass(argc == 2 ? "Password:" : argv[2]);
        encrypted = crypt(pass, pw->pw_passwd);
        if (strcmp(encrypted, pw->pw_passwd))
            exit(1);
        else
            exit(0);
        }
    else
        exit(0);
}
```

Intruder Isolation And Monitoring

Stephen Hansen and Michael Eldredge

Electrical Engineering Computer Facility
Stanford University

ABSTRACT

The Stanford University campus network, SUNET, consists of a single 10 MHz Ethernet spine connecting over 60 subnets via routing gateways. The various subnets provide connections to over 1400 hosts. The mix of hosts include a few large mainframes of various types and several hundred each of DEC VAXes, Suns, IBM PC, and Macintoshes. The majority of the VAXes and Suns are running 4.2 or 4.3 BSD variants and are the target of most security violations. Much of the terminal and modem access to the systems on the SUNET is done through terminal servers called Ethertips (the basis of the current cisco products). These Ethertips can support from 8 to 80 lines, providing telnet access to the hosts. The SUNET is also gatewayed to the ARPANET and NSFNET.

Most of the serious security breakin attempts at Stanford (i.e. the successful ones) are done from modems connected to one of the Ethertips. In particular in late 1986 Stanford was the target and the ARPANET access point for a major series of breakins involving dozens of systems at educational, commercial, governmental, and military sites across the U.S. and even a few in the U.K.

In earlier incidents of this type the intruders were immediately locked out of the systems as soon as they were discovered. In this case when the breakins were discovered the intruders were locked out of all penetrated systems except one. On this system, the newest and fastest one involved, a single account was left open and a system was developed to monitor and log all of the activity on that account. The logging was accomplished by modifying the system's telnet daemon to detect a login to the compromised account and to log the bit stream in both directions. These intruders came onto the SUNET va a single Ethertip connected to a bank of 32 modems connected to a couple of rotories. A set of alarm shell scripts was set to look for a log into this account and either notify the staff directly if they were logged in or to send mail to their accounts if not. When logins were detected the local telephone security people were notified and attempts were made to trace the intruder. (This effort is a major story in itself). In general the intruders used the Stanford systems as a base of operations from which to either directly attempt access to other systems or to down-load password files from their own PC's and run a UNIX password guessing program on stolen /etc/passwd files.

This concept of isolation and monitoring of the intruder rather than immediately attempting a complete lockout has both benefits and risks. The benefits include:

1) The ability to immediately find out which systems are penetrated and to shut out the intruders with some confidence that you've removed any trap-doors or trojan horses.

2) Determine the intruder's techniques and any security holes.

3) Keep the intruders in one place long enough to track them down.

4) Develop sufficient evidence for a successful arrest and prosecution of the intruders.

The major risk of course is that the intruder will cause significant damage to your system or possibly others but this can be minimized. We feel that the benefits outweight the risks in most cases.

HACKMAN: A Systematic Study Of Real Computer Security Holes

Peter Shipley, Russell Brand

TIGHT

ABSTRACT

Computer Security is violatable because of failures both in software and in policy setting. While there are hundreds (if not thousands) of particular tricks that allow one to gain extended privileges, the bulk of these techniques use one of a small number of weaknesses found in Unix systems. By understanding why each of these attacks works, the underlying weakness can be found and corrected. HACKMAN, a book in progress is being written to meet this need.

Motivation:

Without constant vigilance computer security is a phantom. Systems very quickly become insecure if there is not an on going program to keep them secure. Baldwin's work demonstrates that security holes are reintroduced regulary by system maintainers. Research at Stanford has shown how quickly .rhost bugs can be exploited, and other peoples work has shown how many users will choose bad passwords that can be easily determined.

It is well understood that that there are many things that the user must get right every time at user level. Wood and Kochan suggest that there are analogous events that one has to get right every time at the programmer level (e.g. checking the arguments to popen carefully). In the body of the text we will examine a number of weakpoints and trace them back to the fundamental attributes.

Solution Method:

Our approach to solving computer security problems is to explicitly enumerate them with explanation and solutions. A sample entry from our book is given below as well as a partial Table of Contents.

Sample Entry:

3.2. Yp Services

Yp is a network data base server that is used to distribute various system files such as the passwd, hosts, and mail aliases.

Under the vanilla Sun OS distribution the makefile leaves the yp data base files world writeable. All a cracker would need to do to become root is modify the master database files, issue the yppush(8) command, and the passwds become what he wants them to be.

The fix for this is to just add a line to the yp data base makefile that will chmod the file to 644.

Another problem with the Sun Microsystems' yp data base server is that it does not run on a privileged socket, thus any user can "announce" himself to be a yp database server without having to be root.

This permits any user to distribute his own version of the passwd database, by executing their own copy of ypserv(8). This is done by either recompiling or editing the binaries of the ypserv daemon to read from a directory other than /usr/etc/yp.

Any user can do this by copying the current yp data base files into his work directories. He creates his own passwd data base with a root login that has a passwd of his choosing. using the makedbm(8). Then run the modified version of ypserv that uses his database files. The final step is to issue the yp command ypset(8) to point the ypbind daemon at the system running their own version of ypserv. Now all that has to be done is to login, or su, to that system as root.

The Fix

Since this has not been fixed in the current release of yp network data base system I suggest you chmod the executables ypserv, ypset, and ypbind unreadable and non-executable by "other" and non-"wheel" group members.

Another thing you may wish to do to prevent this is to modify the ypserv daemon to run from a privileged port. In addition to having it's clients ignore ypservers that do not use these ports.

Partial TOC:

2.9. Mounting NFS filesystems
2.18. Sun socket problem

3. System level

3.1. Setuid Shell Scripts
3.2. Yp Services
3.3. Privileged sockets from ftp
3.4. Sun Ftp
3.5. Console Output
3.6. Crashing though Sbrk
3.7. Debuging su
3.8. Snarfing passwords
3.9. Crashing Through IPC
3.10. Crashing with ether

4. Procedural Attacks

4.1. The Easiest way
4.2. Decrypting files

5. Source code

5.1. Big files
5.2. Crashing With Sockets
5.3. fakemail.sh
5.4. gpw.sh
5.5. hang.c
5.6. Type
5.7. Set UID files
5.8. xtop.c

Preprint Policy:

As we wish to create as exhaustive a discussion as possible of the Unix security problems and their fixes, we are actively seeking collaboration. Copies of the work in progress can be obtained under written agreement to neither circulate of reproduce HACKMAN prior to its official releases, nor exploit the weakness in a descrutive, illegal, etc manner.

Bibliography:

Software Technical Bulletin, 1988, Sun Microsystems, March 1988. 812-8801-03

Baldwin, Robert W., *Rule Based Analysis of Computer Security*, MIT, 1987.

Brand, Russel, *Brand's Compcon Tutorial*, 1985. UCRL95980

Michael, George and Russel Brand, *MSS Computer Security.*, 7th Annual IEEE conference on Mass Storage Systems., Montery CA..

Morris, Robert and Ken Thompson, *Password Security: A Case History*, AT&T Bell Laboratories, Murray Hill, New Jersey 07974. SMM:18

Reid, Brian, *Reflections on some recent widespread computer breakins*, vol. 30, no. 2, pp. 103-105, Association for Computing Machinery, Feb. 1987.

Ritchie, Dennis M., "On the Security of UNIX," Unix system documention. SMM:17

Shipley, Peter and Russel Brand, *Hackman.* (In Preparation)

Stallman, Richard, *Gnu emacs Source code.* Free Software Fundation, Inc.

Stoll, Clifford, "Stalking the Willy Hacker," *Communications of the ACM*, vol. 31, no. 5, pp. 484-497, Association for Computing Machinery, New York, NY., May 1988.

Wood, P H and S C Kochan, *Unix System Security*, Haydan Book Co., 1985.

Software License Management in a Network Environment

Andrew H. French, Antoinette F. Hershey, Edward J. Wilkens

Computervision

ABSTRACT

CADDStation is a networked platform based on Sun Workstations and Sun OS, which supports a variety of CAD/CAM, Solid Modeling, and Electronics CAE and CAD Products. The workstations form a distributed computer environment coupled by ethernet and many layers of networking software, and arranged as a multiplicity of logical standalone systems or logical server-client clusters. The logical system refers to the delivery of the CAD software. Each logical system may utilize other file servers or device hosts on the network, as well as local graphics accelerators for high performance.

The promise of these networked workstations, that sharing data, applications, and processing power across the network will vastly improve organizational flexibility while retaining single user characteristics of dedicated workstations, is often blocked by the methods which must be used to enforce the licensing of software on the network. Software that uses the most common form of license control, node locking, that is, tying software to a specific CPU, defeats most of the benefits of network computing. On the other hand, site licensing provides maximum network flexibility, but is almost impossible to price in a rational way. A method is needed to securely manage licensed software to allow reliable coupling of cost to usage.

A license is an object that confers the right for any copy of a specific software product to be executed on any CPU. Management of the software is decoupled from management of licenses, and retains the maximum organizational and administrative flexibility. License management inherits the problems of secure coupling of usage to cost.

Maximum security is provided in our solution by the use of small hardware License Managers, coupled to each workstation that may execute licensed software. Whenever a licensed software product is invoked, it queries the License Manager via a software daemon for the presence of an appropriate license. An invocation of the software that finds a license keeps it busy until the software terminates, when the license becomes free for reuse. Thus, multiple users are allowed to share a single license by sequentially acquiring it. Multiple licenses may be used by as many users as there are licenses executing the software simultaneously.

The License Managers communicate over the network by means of the daemons on their local CPUs. While software queries the License Manager on its local CPU, the License Manager can acquire a license across the network. Search of other systems to which it has been granted access through license administration files is carried out automatically if no license is found locally. Security and protection of these license administration files is handled analogously to security and protection of other UNIX administrative files. This aspect of security of licenses is the relationship among users on the network and is not addressed by these methods except as already provided by UNIX.

In addition, explicit movement of licenses between License Managers on different CPUs may be requested by administrators. These explicit requests are also subject to license administration files for permission.

The main aspect of security provided by the hardware License Managers is the preservation of licenses. This controls the relationship between the software vendor and the user community, and is intended to prevent the accidental or purposeful distortion of the coupling of usage to cost. The License Manager possesses an embedded processor and an internal battery-powered RAM that may be loaded with over 1000 individual licenses for as many as 750 distinct software products.

The CPU daemon sends encrypted requests to the License Manager, which replies with an encrypted response which is verified for authenticity by the daemon. All transfers of licenses across the network are encrypted and verified, preventing the "manufacture" of licenses by anyone but an authorized source of licenses. This manufacture of licenses is accomplished by proprietary controlled software that communicates with the License Manager during the manufacturing process. Licenses are not created at any other time, only moved.

This methodology does not rely on the protection or encryption of any files on the workstation disks, and therefore ties nothing, software or files, to any CPU. The License Manager may be physically moved from CPU to CPU. Therefore, no degradation of numbers of licenses occurs when a disk or a CPU is out of service. The License Manager is a low component count, non-mechanical device, yielding a very good MTBF. Therefore, license availability is independent of workstation availability.

Using TUNIS,* A UNIX** Compatible Kernel, as a Basis for Security

M. J. Funkenhauser, R. C. Holt
Computer Systems Research Institute
University of Toronto

ABSTRACT

TUNIS is a UNIX compatible kernel written in Turing Plus. The TUNIS operating system was designed to be a structured system with emphasis on portability and modularity. The structured and modular nature of the TUNIS system lends itself to be a useful base towards building a B3 level secure system. Turing is a general purpose programming language that has a complete formal semantic specification that is suitable for developing mathematical proofs of correctness for programs. Turing Plus is an extension of Turing that includes features suitable for systems development, such as concurrency and mutual exclusion, exception handling and separate compilation.

1. Introduction

TUNIS is a UNIX compatible kernel written in a language, called Turing Plus, that provides modern software engineering features such as modularity, limited visibility (or information hiding), interprocess communication and concurrency. The original goal of TUNIS was to build a large system that was structured, modular and portable.

The Secure TUNIS project was started when it was discovered that the design principles used in the development of TUNIS satisfy many of the recommendations of the TCSEC[1]. We believe that the TUNIS structure combined with the Turing Plus language, in which TUNIS is written in, are particularly appropriate for implementing a secure system. Turing Plus is a strongly typed, formally specified language which features modularity, interprocess communication and concurrency. These features go a long way towards providing assurances necessary in any trusted system development.

---

* TUNIS is a registered trademark of the University of Toronto

** UNIX is a registered trademark of AT&T in the USA and other countries.

[1] Department of Defense Trusted Computer System Evaluation Criteria, DOD 5200.28-STD, December 1985.

2. TUNIS History

TUNIS (Toronto UNIversity System) evolved from teaching about operating systems at the University of Toronto. A graduate course designed the first version in 1979 and in 1980 a more ambitious UNIX compatible system was attempted. A Version 7 compatible system running on a PDP-11/23 was completed in 1982. In 1984 the system was successfully ported to the National Semiconductor 32000 architecture. During the port, the system was modified to be 4.1 BSD compatible and paging virtual memory management was added. Since 1984 several multi-processor projects have been completed including master/slave/real-time TUNIS [BLYTHE85] and symmetric multi-processor TUNIS [EWENS85]. Current projects involving TUNIS include HECTOR (100 processor TUNIS) and Secure TUNIS.

3. TUNIS Design Goals

The TUNIS system is designed to adhere to the major principles of structured software design including modularity, portability and ease of understanding. One purpose of the TUNIS project has been to develop high level software structuring techniques in a large system; techniques including strong type checking, concurrency and information hiding.

The principle vehicle for attaining these achievements is the implementation language - Turing Plus (T+). T+ provides enforced modularity and encapsulation through its *module* construct and this provides the basic building block for the internal structure of TUNIS. Each module in TUNIS implements an abstraction by containing a set of data and a set of *entry points* that allow operations to occur on the data.

Each module's entry points are designed to achieve limited visibility. That is, the details of the algorithms and data structures inside each module are not visible to other modules. This allows modules to be changed independently of one another as long as the interface between modules is maintained. A notable example of this occurred in practice when only one module was replaced to switch from swapping to paging based memory management.

The design of a module interface involves a trade off between the conflicting goals of limited visibility and high performance. To achieve better performance in certain areas, module-to-module interfaces are widened to provide specialized routines for frequently used operations.

Use of *textual isolation* within modules improves readability and isolate machine dependencies. For instance, sub-modules are used to localize the code that performs physical I/O and other sub-modules are used to contain configuration dependent data declarations.

4. TUNIS Structure

The internal structure of TUNIS is heavily influenced by the features of Turing Plus (T+), such as modules, monitors and processes. Processes are used to support concurrent activity and monitors are used to gain mutually exclusive access to data. T+ processes allow the TUNIS system to manage the concurrency of asynchronous activities, such as devices and user processes. Each asynchronous activity

represents a *thread of control* within TUNIS and each of these activities corresponds to a T+ process.

The following is a brief summary of the various modules within TUNIS but a more detailed look at the internal structure can be found in [CSRI86].

The entire TUNIS system is composed of a hierarchy of seven major modules: the User manager, the File manager, the Memory manager, the Device manager, the Clock manager, the Panic manager and the Concurrency Kernel[2]. Each of these managers/modules implements a new level of abstraction. Control within TUNIS is passed strictly hierarchically between the modules such that a module can call lower level modules but never a higher level module.

The User manager, which is the top most module, implements the interface between user programs and the TUNIS nucleus. It contains a set of T+ processes called *envelopes*. Each envelope supervises a user process by repeatedly performing system services (system calls) requested by the user process until the user process terminates. Envelopes are queued in a sub-module, called the Family manager, waiting for a user process to be "born" via a *fork*. An envelope is dispatched to start and then supervise the new user process. The envelope loops until the user program terminates, alternating between running the user and servicing the user's system calls. The envelope may call on other TUNIS modules to perform the requested system call.

The Family manager maintains the hierarchy of user processes (e.g., parents, childrens, grandchildrens) and is responsible for user interprocess communication. It regulates how children (user) processes are created, notifies the parent (user) process when the child terminates, and keeps track of signals pending on each user process.

The File manager implements the UNIX file system by maintaining the hierarchical file system structure and enforcing file protections. The File manager is completely machine and device independent. Within the File manager are four major sub-modules: the FlatFile manager, the TreeFile manager, the Inode manager and the Cache manager.

The TreeFile manager implements the hierarchical or tree file system consisting of directories as internal nodes and non-directory files as leaves. The manager's function is to create and delete new files in the hierarchy and to find the canonical identity of existing files in the hierarchy (i.e., to perform path name translations).

The FlatFile manager maintains information about files that are open to the system and open to user programs. It is also responsible for the management of UNIX special files, such as pipes, and the routing of I/O requests to the Inode manager.

The Inode Manager is responsible for the physical layout of files on disks. It handles the creation, deletion and dynamic expansion of disk files. From the Inode manager, all requests for file I/O are passed to the Cache manager.

---

[2] In UNIX, the low-level operating system is referred to as the kernel. In TUNIS, however, we refer to the operating system as the nucleus since within the TUNIS nucleus there is a low level support module called the Concurrency Kernel.

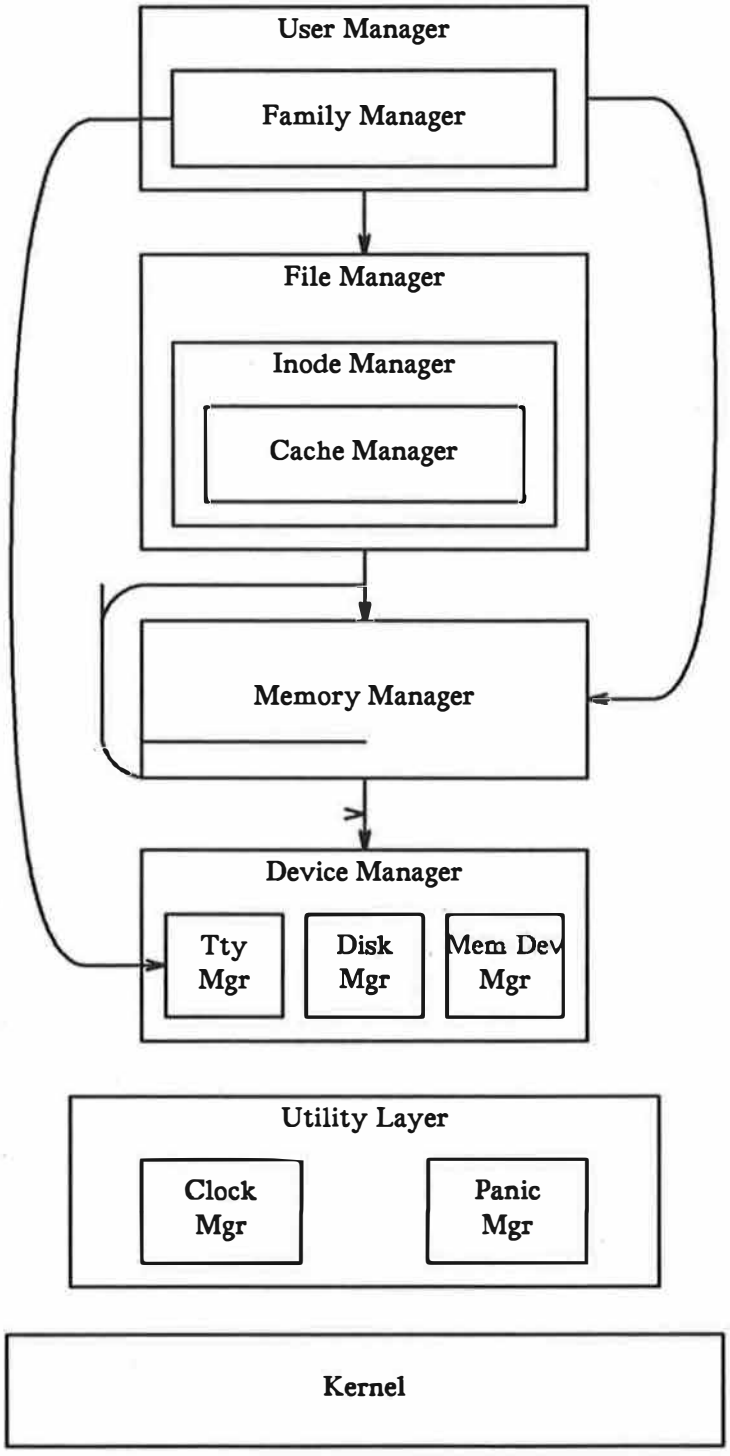

Figure 1. Structure of the TUNIS Operating System

The Cache Manager is responsible for buffering I/O requests to all devices. It contains a pool of buffers that are used to keep copies of recently accessed device blocks such that repeated access to the same device block will just access the in-core buffers instead of the block on the physical device.

The Memory manager schedules and allocates memory resources which include physical memory and temporary disk space (i.e., swap area). This manager implements the abstraction of virtual memory for each user process, providing memory protection for both the user's and the system's address spaces. Only the Memory manager knows the exact locations of user programs. The Memory manager supports operations to expand, shrink, duplicate and destroy partitions as well as the transfer of data between partitions.

The Device manager is the interface between the higher levels of TUNIS and the hardware devices. This manager contains sub-modules including the Tty manager and the Disk manager.

The Clock manager keeps track of the time of day and provides an interface that allows processes to be delayed (to sleep) for a specified amount of time.

The Panic manager is responsible for the handling of error messages. This includes printing the messages on the console and to buffer the messages in a pseudo-device which a user process can read and copy the messages into an error message log file.

At the bottom of the TUNIS hierarchy is a module called the concurrency Kernel which is inherently machine dependent. Its performance is critical to the overall performance of the system. For these reasons it is the only part of TUNIS that is partially written in assembly language. The Kernel module can be thought of as an extension of the hardware to support the concurrency constructs of T+. The primary responsibilities of the Kernel are:

- implement concurrency via T+ processes,
- enforce mutual exclusion within monitors,
- handle interrupts,
- provide a simple mechanism for interfacing to peripherals,
- passing trap codes from user process to TUNIS nucleus (i.e., system calls),
- dispatching user processes.

5. Turing and Turing Plus

Turing [HOLT83b], with its extension to Turing Plus [HOLT85], is a general purpose programming language that is designed to have Basic's clean interactive syntax, Pascal's elegance and C's flexibility. Turing and Turing Plus is the successor to Concurrent Euclid (CE) [HOLT83a].

The definition of the Turing language includes a formal mathematical specification. This definition strives to maximize the degree that the language is defined in terms of mathematics. The goals for the formalization of Turing are *completeness*, *consistency* and *clarity*. Completeness means that all relevant aspects of the language are defined. Consistency means that there is only one way to

interpret the definition. Clarity means that the definition is easy for a person to read and understand.

There are several good reasons for providing a formal definition of the language. The foremost reason is that mathematical notation provides a degree of precision that simply is not possible in natural language descriptions. A second reason is to assist the implementor of the language. The formal semantics of the language serve as a guideline to a programmer implementing the compiler for the language. A third reason is to assist the Turing programmer. Turing's formal semantics can serve as a basis for either proving programs correct or for methodologies for developing correct programs.

Turing is a *puritanical* language in that it is completely defined mathematically and, given that checking is in force, it is not possible for Turing programs to directly access the underlying implementation. Dangerous features, such as C's pointer arithmetic and type cheating have been eliminated by careful language design. All features of Turing are *clean* meaning that their semantics are mathematically defined. They are also *faithful*, meaning that there are run-time checks to guarantee that an executing program never violates the run-time constraints that would invalidate the formal semantics.

Turing Plus is a compatible extension of Turing. The extensions allow a controlled access to essentially all aspects of the run-time implementation. In other words, Turing Plus is a *permissive* language. Turing Plus provides the programmer with flexible, machine oriented features to allow systems programming without sacrificing the checking and elegance of the core features of the language. These features are considered to be *dirty* in that the meaning of the features are inherently implementation dependent. There are three increasingly machine-dependent levels of access to the underlying implementation:

- data visibility - access to bits and bytes
- address visibility - access to addressing of variables
- code visibility - access and control of emitted machine language

6. Turing Plus Features

Integers (*int*), reals (*real*) and strings (*string*), which are among Turing's basic data types, are extended to include natural numbers (*nat*), addresses (*addressint*) and characters (*char*). Extensions have also been made to allow the programmer explicit control of data sizes (i.e., *int1*, *int2*, *int4*, *nat1*, *nat2*, *nat4*, *real4*, *real8*).

New operations have been added such as bit manipulation, type cheats, arbitrary manipulation of machine addresses, concurrency and interprocess communication.

The basic set of concurrency and interprocess communication features are clean (implementation independent) and are suitable for portability of programs across multi-CPU configurations. These features include *processes*, *monitors*, *condition variables* and *signal* and *wait* primitives that operate on condition variables.

Processes in Turing Plus are "true processes", meaning that they can physically run in parallel if multiple CPUs are available. Interprocess communication is

provided by Hoare style monitors which provide mutually exclusive access to shared data as well as mechanisms for blocking and awakening of processes.

7. Secure TUNIS

Secure TUNIS is a new project whose objective is to make TUNIS conform to the IEEE POSIX System Interface standard as much as possible with additional security mechanisms that will satisfy the functional criteria for a B3 level system as stated in [DOD85]. It is believed that the modularity and structure of TUNIS combined with the Turing programming language, which has been formally specified, is the ideal starting platform for a B3 level UNIX like system.

We are currently in the process of finalizing a security policy that can be incorporated into TUNIS and still retain most POSIX compatibility. During the past two years there have been several research projects related to TUNIS and security. One project dealt with the problem of how to perform formal specification of the TUNIS software [GODFREY88] and the project was concerned with the reference monitor concept and how TUNIS could be restructured to implement this concept [GRENIER88].

References

[BLYTHE85] Blythe, D.R., "Master/Slave TUNIS: A Low-Cost, High-Performance Multiprocessor Operating System", *MSc Thesis Report*, Department of Computer Science, University of Toronto, September 1985.

[CORDY81] Cordy, J.R., Holt, R.C., "Specification of Concurrent Euclid", *Technical Report CSRI-133*, Computer Systems Research Institute, University of Toronto, August 1981.

[CSRI86] Ewens, P.A., Holt, R.C., Funkenhauser, M.J., Blythe, D.R., "The TUNIS Report: Design of a UNIX Compatible Operating System", *Technical Report CSRI-176*, Computer Systems Research Institute, University of Toronto, January 1986.

[DOD85] U.S. Department of Defense, "Department of Defense Trusted Computer System Evaluation Criteria", DOD 5200.28-STD, December 1985.

[EWENS85] Ewens, P.A., "Symmetric TUNIS: A Multiprocessor Operating System", *MASc Thesis Report*, Department of Electrical Engineering, University of Toronto, September 1985.

[GODFREY88] Godfrey, M.W., "Toward Formal Specification of Operating System Modules", *MSc Thesis Report*, Department of Computer Science, University of Toronto, May 1988.

[GRENIER88] Grenier, G.-L., "Toward Secure Tunis, A Trusted Operating System", *MASc Thesis Report*, Department of Electrical Engineering, University of Toronto, expected July 1988.

[HOLT83a] Holt, R.C., "Concurrent Euclid, the UNIX system, and TUNIS", Addison-Wesley Publishing Co., Reading, Massachusetts, 1983.

[HOLT83b] Holt, R.C., Cordy, J.R., "The TURING Language Report", *Technical Report CSRI-153*, Computer Systems Research Institute, University of Toronto, December 1983, revised 1985, updated 1986.

[HOLT85] Holt, R.C., Cordy, J.R., "The TURING Plus Report", *Technical Memo*, Computer Systems Research Institute, University of Toronto, March 1985, revised 1986, revised 1987.

[HOLT87] Holt, R.C., Mathews, P.A., Rosselet, J.A., "The TURING Programming Language: Design and Definition", Prentice-Hall Publishing Co., Englewood Cliffs, N.J., 1987.

Suggested Levels of Security

Rick Lindsley

Tektronix

It is difficult to gauge the security of a system. There are some common things to look for, however, and the presence or absence of these items can be a general indication of the level of security. This paper attempts to provide guidelines as a means of comparison, and to give system owners a goal to strive for.

Minimal security would be recommended for any system which is being used for serious business work, or may contain data related to serious business work. "Serious business work" is defined rather vaguely as any work which may be beneficial to business owning the machine. This is intentionally broad, as it is recommended that no system in use for business, perhaps excepting those used as "crash" machine, should fail to meet the criteria listed here.

The sensitivity of the work being done or the data being stored should dictate to the system owner the level of security desired for his or her system.

These levels do not correlate to any federal security requirements, as outlined in "The Orange Book." Requirements for universities may differ from these, as a university environment may differ considerably. Nonetheless, a university which can attain level 1 as listed here could sleep more easily at night. These levels are primarily to evoke discussion and cause system administrators to think about what level of security *they* require for *their* site(s).

They proceed from relatively easy-to-meet criteria to a very tightly controlled environment. As restrictions tighten, user freedom decreases. This may evoke howls of protest from the users, so the sysadmin must balance the overall need for security with the needs of his or her users.

Level 1 — Minimal Security

Physical Security

CPU The CPU should have access restricted during non-business hours; for example, having the CPU in a room or area which is locked after hours but left open during normal business hours would satisfy this requirement.

Disks All disks containing data used by the machine (including, but not limited to, floppies or removable hard disks) should be under the same restrictions as the CPU, although disks and CPU need not be stored together. If the disks are physically small, removable, and easily stored, as with floppies, they should be in a locked drawer or cabinet during non-business hours.

Administrative Security

- All accounts on the machine should have passwords.

Level 2 — Low Security

All requirements for Level 1 must be met. In addition:

Physical Security

TapesTapes which contain data from the indicated system should have the same security as disks.

Terminals

The physical location of all hard-line terminals should be documented, as well as which ports connect to which terminals. If the system connects to one or more dataswitches, those ports on the computer which are used by the dataswitch(es) should also be documented. [This information can be useful in determining where an intruder has gained entry.]

Administrative Security

- Passwords shall be checked for easily guessed passwords at least once every 3 months. "Easily guessed passwords", for the purpose of this document, shall be:

  the same as the login
  the user's first name
  the user's last name

  Note that more stringent checking is possible, and encouraged; however this is all that is *required* at this level.

- New accounts shall not be given easily guessed passwords by default. Ideally, the user should verbally tell the account administrator the desired password, so that a legitimate password is used, but not written down anywhere (at least, not by the account administrator.)

Level 3 — Moderate Security

All requirements for Level 2 must be met. In addition:

Physical Security

CPU Access to the CPU shall be restricted at all times. A list of people with access to the restricted area should be available to auditors.

DisksAccess to disks shall be restricted as with the CPU.

TapesAccess to tapes shall be restricted as with the CPU.

Terminals

Dialin access, if available on this machine, should be restricted in some fashion. This can be done in a variety of ways, such as keeping the phone numbers known to a restricted list of people, or instituting double passwords for dialin lines (one for dialin access, one as part of a normal login procedure).

If a dataswitch is used, access should also be restricted to the degree that this is possible. Sytek, for instance, can have non-published addresses. A person may *guess* an address and find a machine, but otherwise cannot find it except from another person who knows it.

Administrative Security

- Password aging of some sort shall be in effect, forcing passwords to be changed at least once every six months.
- Old accounts shall be deleted promptly — within two weeks of their being designated unneeded or abandoned.

Level 4 — High Security

All requirements for Level 3 must be met. In addition:

Physical Security

Terminals

The computer shall not have dialin capability. All access must be guaranteed to be only from inside the business. Dataswitches are permitted, if they themselves can guarantee business-only access (no dialin capability.)

Administrative Security

- Password aging shall require that passwords be changed a minimum of every two months.
- The system owner shall review all accounts at least once every three months and verify that a) each account is still needed and b) is still being used. It is recommended, though not required, that accounts which are inactive but which contain potentially useful information be archived (so that the information is not lost) and then removed.

Level 5 — Extremely High Security

All requirements for Level 4 must be met. In addition:

Physical Security

CPU All accesses to the location where the CPU is secured shall be recorded. Card keys (when card keys are the *only* access) can provide this level of security.

DisksSame as CPU.

TapesSame as CPU.

Terminals

There shall be no dataswitch access. Physical integrity of lines leading to terminals should be inspected at regular intervals to insure that no "tapping" of the lines is being done.

Network

There shall be no network access (through an Ethernet, Hyperchannel, or similar device) unless:

- The network logically connects to 10 or fewer hosts
- All hosts reachable on the network are *also* Level 5

Level 6 — Paranoia

All requirements for Level 5 must be met. In addition:

Physical Security

Terminals

All access to the machine shall be from a restricted area. An example of this is a terminal room secured by a lock, with the room always locked, and a limited number of people possessing a key, combination, or appropriate card key authorization.

Network

There shall be no network access; that is, no other computer can contact a Level 6 computer. Any data transfer must be done by physical media such as tape or disk (or entered by keyboard).

Identifying Security Concerns

Rick Lindsley, Seth Alford, Richard Kurschner, Roger Southwick

Tektronix

ABSTRACT

The term "system security" is a rather general and all-encompassing term. When a group of concerned administrators and programmers at Tektronix decided to get together and *do* something about it, we realized there were three different levels of expertise that we needed to address. Consequently we came up with three different security guides, written in terms that each audience should understand.

- **The user level.** Security begins with the users. Naive users are far more common and likely to leave themselves subject to security problems than a sophisticated user. Hostile users probably won't stop because you asked them to, but naive users can be educated.

  Topics covered in the *A User's Guide to UNIX Computer Security* are

  - the importance of passwords
  - keeping your terminal secure
  - keeping your files secure
  - security over the network
  - how to detect when your account has been broken into
  - the importance of securing your personal archives too (tapes and such)

- **The administrator level.** The administrator is the first (and frequently the last) line of defense against intruders. He or she must be prepared to foil attempts at external breaches (dial-ins), internal breaches (nosy or hostile users) and even physical breaches (physical access to the computer.) It is no surprise, then, that this became the largest of the three documents prepared.

  It takes a rather paranoid look at the possible ways a machine could be compromised. It was decided it was better to look at things from a worst-case point of view, and then say it is up to the administrator to do that much or less, than to discuss less and leave the administrator to figure out what more he or she could do.

  Topics covered in *UNIX System Security for System Administrators at Tektronix* include:

  - what information you may want protected
  - precautions to take while root (to avoid traps)
  - suggested scripts and actions you may take to detect breakins
  - suggestions on how to physically protect your machine

- the level of security possible on an ethernet
- what to do when you've detected a breakin

- **The owner level.** Frequently in business, and to a lesser extent in university situations, the person who actually owns the machine does not understand the responsibility inherent in that. This guide attempts to convey to the owner why this is important and steps he or she may wish to take. Many of these steps might well be relegated to the system administrator, if such a person exists, and so the owner's guide and the administrator's guide overlap somewhat. However, the owner's guide is far less technical and much more "why" than "how."

  Topics in *UNIX System Security for System Owners at Tektronix* include:

  - reasons for protecting data
  - criteria for determining what level of security a owner may wish to enforce on a machine
  - the importance of passwords
  - networks and security
  - physical security

These three documents have become the basis for much of the effort at Unix security within Tektronix. Sometimes readers are surprised at what they had left themselves open to. System administrators, in particular, have grown more vigilant and expressed a determination to monitor these sorts of things more closely.

Computer Security Measures at Eastman Kodak, Product Software Engineering Department

Ken Lester

Eastman Kodak Company

ABSTRACT

Environment

Our current department environment consists of several computers residing upon a corporate network (ethernet). This network contains thousands of computers, generally running some version of UNIX† or VMS™. Our department computers include a MicroVax II, Vax 11/750, some Sun-2s and many Sun-3s. These computers run Ultrix™ or SunOS versions of UNIX. All computers support NFS, and several computers export file systems to other computers. The Vax 11/750 has modems with restricted dial-in capabilities. The Vax 11/750 can also be accessed via a broadband LAN.

Security Solutions

Our initial security concern was protecting our computers that have dial-in connections. One facet of the solution was the creation of a program called *aclogin*, which is invoked by *getty*. When *aclogin* is invoked, it reads a special file that consists of an user name field, access code field, and 1 or more fields listing devices which the user can log onto (the list can be regular expressions). If the user has permission to log into the device, *aclogin* requests an *access code*, which can be thought of as another password. If the access code is correct, then *aclogin* invokes *login* to complete the process. Several points worth noting:

1) The reason *aclogin* invokes *login* is because we don't have source code to determine everything *login* does.

2) A small subset of users have modem access.

3) The *aclogin* file is only readable by root. This makes it difficult for anyone to know the list of users who can access the modems. We were concerned that if such a list is readable, then it is easier for an intruder to obtain the list, and thus try intelligent guesses at the access codes and passwords.

The advent of NFS has caused us concerns. Current implementations are not very secure. We have investigated possible ways for intruders to use NSF for illegal access. Some of the less obvious security steps include:

1) We explicitly list equivalent hosts. We have found that many sites will have a line in */etc/hosts.equiv* consisting of an operator "+", which allows any host to be equivalent. This is a common administrator mistake, especially with systems having Yellow Pages which uses this operator in

† UNIX is a trademark of Bell Laboratories.
Ultrix and VMS are trademarks of Digital Equipment Corporation

/etc/passwd and */etc/hosts*.

2) All exported file systems listed in */etc/exports* have corresponding lists of hosts. Not having a host list with a exported file system will allow any host to mount that file system. This would allow a hostile user, with root privileges on a remote host to mount the exported file system and modify and in some cases remove files (note: many NFS implementations will thwart this action for files owned by root). This intrusion can easily be accomplished by the hostile user, who need only add entries in the remote host's */etc/passwd* file to contain *uids* used in the exported file system, and then *su* to these uids.

3) We never export the root file system or any partition containing important and confidential information. This is because steps 1 and 2 are vulnerable to a hostile user with root privileges on a remote host, who happens to change the remote host's name or internet address to one of our equivalent of importing hosts.

We take normal precautions, such as educating our user community on what makes a good password. We also perform some system checking such as checking for new appearances of suid root programs or modifications of such programs. We are concerned about security, and wish to provide our user community with a safe working environment.

UNIX Security at Pacific Bell

Jerry M. Carlin

Pacific Bell

ABSTRACT

Recently Pacific Bell chartered a UNIX Security Task Force as a result of some well publicized incidents. The charter was to dramatically increase UNIX Security by any means necessary within a three month period. A review of existing practices found inadequate awareness, standards, training and communication with vendors. Elementary practices were ignored. Known security problems fixed in later UNIX releases were not communicated to us leaving some systems vulnerable. In response to this a Standards Manual was written, netnews articles published, an internal security mailing list established, several programs were fixed at key systems, contacts were established at key vendors and a number of security breaches were investigated.

Existing Situation

At the start of the effort, the awareness and implementation of security measures was spotty at best. About the only guaranteed security measure was giving "root" a password. Some systems had */etc* mode **777**. Others had similarly bad problems. In one case we found a trojan horse *su* program that was totally unnecessary since the system was so wide open becoming root was trivial. On the other hand, there were a few systems with a high degree of security including special versions of *login* designed to limit number of retries and a special version of *su* that allowed only certain "uid's" to become "root."

Standards Manual

One problem was a lack of knowledge about what constitutes a reasonable set of security measures. In addition, none of the existing resources including published books and papers was found to be complete enough. In response to this need a manual was written. Major sections included administration, auditing, application design and programming, user security, network security and usenet/source/binaries.

Automated Tools

Early on we realized that manually auditing a system was very time consuming. Several people had started trying to enter and modify the scripts from "UNIX System Security" by Wood and Kochan. Others had written some shell scripts for particular environments and particular problems. Part of the standards effort was to gather a full set of requirements for automated tools. A start has been made on writing such a tool set, partially because there is no commercially available complete enough set of tools.

Education

Education is vital in making sure that we are more secure. Until we have people security conscious and taking a starter set of precautions being concerned about esoteric operating system problems is irrelevant. We have to convince people that setting mode 777 on their $HOME and *.profile* is not a good idea. This is being pursued in a number of ways. All members of the task force spread the word that security was important. A small number of people can do a lot by word-of-mouth. Netnews and paper articles help in telling people what to do. A security course will be established to train system administrators on security procedures.

NOTES

USENIX Association Membership Application

Membership is by Calendar Year

Please type or print

[ ] New [ ] Renewal

Name: _____

Address: _____

_____

_____

_____

Phone: _____

uucp network address: *uunet!* _____

Individual, Corporate, and Supporting categories are all open to either institutions or individuals. Membership fees are:

[ ] $ 40 Individual

[ ] $ 15 Student (full-time)
With copy of student I.D. card

[ ] $ 275 Corporate

[ ] $125 Educational Institution

[ ] $1000 Supporting

[ ] Check enclosed: $ _____
Payments must be in US dollars payable on a US bank

[ ] Purchase order enclosed; invoice required

[ ] Check if you do **NOT** want your name and address made available to other members.

[ ] Check if you do **NOT** want your name and address made available for commercial mailings.

Please complete and return this form with your purchase order or payment to:

USENIX Association
P. O. Box 2299
Berkeley, CA 94710

For Office Use

Inst: .....

Mem#: ..... Check#: .....

Lic: ..... Rf: .....

Date: ..... Db: .....

USENIX Association
P.O. Box 2299
Berkeley, CA 94710